International Software Testing Qualifications Board

テスト技術者資格制度

Foundation Level Specialist シラバス
性能テスト担当者

Version 2018.J01

Provided by

American Software Testing Qualifications Board

and

German Testing Board

性能テストワーキンググループ:

Graham Bath

Rex Black

Alex Podelko

Andrew Pollner

Randy Rice

改訂履歴

バージョン	日付	備考
Alpha V04	2016 年 12 月 13 日	NYC ミーティング用バージョン
Alpha V05	2016 年 12 月 18 日	NYC ミーティング終了後
Alpha V06	2016 年 12 月 23 日	4 章のリストラクチャー NYC で合意された LO のリナンバリングと調整
Alpha V07	2016 年 12 月 31 日	作者コメントの追加
Alpha V08	2017 年 2 月 12 日	プレアルファ版バージョン
Alpha V09	2017 年 4 月 16 日	プレアルファ版バージョン
Alpha Review V10	2017 年 6 月 28 日	アルファ版レビュー用
V2017 v1	2017 年 11 月 27 日	アルファ版レビュー用
V2017 v2	2017 年 12 月 15 日	アルファ版のアップデート
V2017 v3	2018 年 1 月 15 日	技術的な修正
V2017 v4	2018 年 1 月 23 日	用語集レビュー
V2018 b1	2018 年 3 月 1 日	ISTQB®のベータ版候補
V2018 b2	2018 年 5 月 17 日	ISTQB®のベータ版リリース
V2018 b3	2018 年 8 月 25 日	ベータ版レビューのコメントをリリース版に反映
Version 2018	2018 年 12 月 9 日	ISTQB® GA リリース

JSTQB®

バージョン	日付	備考
Version 2018.J01	2022 年 4 月 16 日	Version 2018 の日本語翻訳版

目次

謝辞

この文書は、米国ソフトウェアテスト資格認定委員会（ASTQB）とドイツテスト委員会（GTB）によって作成された。

Graham Bath (GTB, Working Group co-chair)
Rex Black
Alexander Podelko (CMG)
Andrew Pollner (ASTQB, Working Group co-chair)
Randy Rice

コアチームは、レビューチームからの提案と意見に感謝する。ASTQB は、このシラバスの開発に貢献した Computer Measurement Group (CMG)を認め、感謝する。

次のメンバーが本シラバス、またはその前身のレビュー、コメント、投票に参加した。

Dani Almog
Sebastian Chece
Todd DeCapua
Wim Decoutere
Frans Dijkman
Jiangru Fu
Matthias Hamburg
Ágota Horváth
Mieke Jungebload
Beata Karpinska
Gábor Ladányi
Kai Lepler
Ine Lutterman
Marek Majernik
Stefan Massonet
Judy McKay
Gary Mogyorodi
Joern Muenzel
Petr Neugebauer
Ingvar Nordström
Meile Posthuma
Michaël Pilaeten
Filip Rechtoris
Adam Roman
Dirk Schweier
Marcus Seyfert
Péter Sótér
Michael Stahl
Jan Stiller
Federico Toledo
Andrea Szabó
Yaron Tsubery
Stephanie Ulrich
Mohit Verma
Armin Wachter
Huaiwen Yang
Ting Yang

本ドキュメントは、2018 年 12 月 9 日に ISTQB®から正式にリリースされた。

日本語訳については、以下の日本語翻訳ワーキンググループメンバーにより行われた。

日本語翻訳ワーキンググループメンバー：
佐々木健旨（日本アイ・ビー・エム株式会社）
湯本剛（株式会社 ytte Lab／JSTQB 技術委員）

0 イントロダクション

0.1 本シラバスの目的

本シラバスは、テスト技術者資格 性能テスト担当者 Foundation Level のベースとなる。ASTQB®と GTB®は、このシラバスを以下の趣旨で提供する。

1. 各国の委員会に対し、各国語への翻訳および教育機関の認定の目的で提供する。各国の委員会は、本シラバスを各国語の必要性に合わせて調整し、出版事情に合わせてリファレンスを修正することができる。
2. 試験委員会に対し、シラバスの学習の目的に合わせ、各国語で試験問題を作成する目的で提供する。
3. 教育機関に対し、コースウェアを作成し、適切な教育方法を確定できるようにする目的で提供する。
4. 受験志願者に対し、試験準備（研修コースの一部、または独立した形）の目的で提供する。
5. 国際的なソフトウェアおよびシステムエンジニアリングのコミュニティに対し、ソフトウェアやシステムをテストする技能の向上を目的とする他、書籍や記事を執筆する際の参考として提供する。

ASTQB®と GTB®では、事前に書面による申請があった場合に限り、第三者がこのシラバスを先に定めた以外の目的での使用を許諾することがある。

0.2 認定 Foundation Level 性能テスト

本 Foundation Level の資格は、ソフトウェアテストに携わる方で、性能テストに関する知識を広げたい方や、性能テストの専門家としてキャリアをスタートさせたい方を対象としている。また、性能エンジニアリングに携わる方で、性能テストについての理解を深めたい方も対象としている。

このシラバスでは、性能テストの以下の主要な側面を考慮している。

- 技術的側面
- 手法ベースの側面
- 組織的側面

ISTQB® Advanced Level Technical Test Analyst シラバス[ISTQB_ALTTA_SYL]に記載されている性能テストに関する情報は、このシラバスと整合しており、このシラバスによって展開されている。

0.3 ビジネス成果

この節では、Foundation Level 性能テスト認定資格を取得した受験者に期待されるビジネス成果を示す。

PTFL-1 性能効率性と性能テストの基本的な概念を理解する。
PTFL-2 ステークホルダーのニーズと期待に応えるために性能リスク、ゴール、要件を定義する。
PTFL-3 性能メトリクスとその収集方法を理解する。
PTFL-4 ゴールと要件を達成するために性能テスト計画書を作成する。
PTFL-5 基本的な性能テストケースを概念的に設計し、実装し、実行する。
PTFL-6 性能テストの結果を分析し、さまざまなステークホルダーへの影響を述べる。
PTFL-7 性能テストのプロセス、理論的根拠、結果、および影響をさまざまなステークホルダーに説明する。
PTFL-8 性能ツールのカテゴリーと用途、およびその選択基準を理解する。
PTFL-9 性能テストの活動をソフトウェアライフサイクルとどのように整合させるかを判断する。

0.4 試験のための学習の目的と知識レベル

学習の目的は、ビジネス成果をサポートし、Foundation Level の性能テスト認定を達成するための試験を作成するために使用される。学習の目的は、知識の認知レベル（K-レベル）に割り当てられている。

K-レベルまたは認知レベルは、Bloom（ブルーム）[Anderson01]の改訂版分類法に基づいて学習の目的を分類するために使用される。ISTQB®では、この分類法を用いてシラバス試験の設計を行っている。

このシラバスでは、4 つの異なる K-レベル（K1～K4）を考慮している。

K-レベル	キーワード	説明
1	記憶	受験者は、用語や概念を覚えたり、認識したりする必要がある。

2	理解	受験者は、質問項目に関連する記述の説明を選択する必要がある。
3	適用	受験者は、概念や技術の正しい適用を選択し、与えられた状況に適用する必要がある。
4	分析	受験者は、手順や技術に関連する情報をより理解しやすいように構成要素に分け、事実と推論を区別することができる。

基本的に、本シラバスのすべての箇所は、K1 レベルで試験対象となる。つまり、受験志願者は用語や概念について認識し、記憶し、想起することになる。関連する章の最初には K2、K3、K4 レベルの学習の目的のみを記載する。

0.5 推奨トレーニング時間

このシラバスでは、学習の目的ごとに最低トレーニング時間が定められている。各章の総所要時間は、章の見出しに記載されている。

教育機関は、他の ISTQB®シラバスが、K-レベルに応じて固定時間を割り当てる「標準時間」アプローチを適用していることに留意する必要がある。性能テストのシラバスは、この方式を厳密には採用していない。その結果、教育機関は、各学習の目的のための最小トレーニング時間をより柔軟かつ現実的に示すことができる。

0.6 認定資格試験の受験資格

Foundation Level 性能テストの認定資格試験を受験するには、Certified Tester Foundation Level 認定資格を取得している必要がある 。

0.7 情報源

シラバスで使用される用語は、ISTQB®の「ソフトウェアテストで使用される用語集」[ISTQB_GLOSSARY]で定義されている。

6 章では、性能テストに関する参考書籍や記事を紹介している。

1. 基本的な概念 – 60 分

キーワード

キャパシティテスト、コンカレンシーテスト効率性、耐久テスト、負荷生成、負荷テスト、性能テスト、拡張性テスト、スパイクテスト、ストレステスト

「基本的な概念」の学習の目的

1.1 原則と概念

PTFL-1.1.1 (K2) 性能テストの原則を理解する。

1.2 性能テストのテストタイプ

PTFL-1.2.1 (K2) 性能テストのテストタイプを理解する。

1.3 性能テストにおけるテストの種類

PTFL-1.3.1 (K1) 性能テストにおけるテストの種類を想起する。

1.4 負荷生成の概念

PTFL-1.4.1 (K2) 負荷生成の概念を理解する。

1.5 性能テストにおける一般的な故障とその原因

PTFL-1.5.1 (K2) 性能テストの一般的な故障モードとその原因の例を挙げることができる。

1.1 性能テストの原則

性能効率性（または単に「性能」）は、固定およびモバイルのさまざまなプラットフォームでアプリケーションを使用する際に、ユーザーに「よい体験」を提供するために不可欠な要素である。性能テストは、エンドユーザーに受け入れられる品質レベルを確立する上で重要な役割を果たしており、多くの場合、使用性エンジニアリングや性能エンジニアリングなどの他の分野と密接に統合する。

また、性能テストの実施時など、負荷がかかった状態で機能適合性や使用性などの品質特性を評価することで、それらの特性に影響を与える負荷固有の問題が明らかになることがある。

性能テストは、エンドユーザーに焦点が当たる Web ベースの領域に限定されない。従来のクライアントサーバー型、分散型、組込み型など、さまざまなシステムアーキテクチャーを持つさまざまなアプリケーションドメインにも関連する。
技術的には、性能効率性は、ISO 25010 [ISO25000] プロダクト品質モデルにおいて、以下の 3 つの副特性を持つ非機能品質特性に分類される。適切な焦点と優先順位付けは、評価されたリスクとさまざまなステークホルダーのニーズに依存する。テスト結果の分析により、対処すべき他のリスク領域が特定される場合がある。

時間効率性：時間効率性の評価は、最も一般的な性能テストの目的である。性能テストのこの側面では、コンポーネントやシステムが、指定された時間内に、指定された条件下で、ユーザーやシステムの入力に応答する能力を調査する。時間効率性の測定値は、システムがユーザーの入力に応答するまでの「エンドツーエンド」の時間から、ソフトウェアコンポーネントが特定のタスクを実行するのに必要な CPU サイクル数まで、さまざまである。

資源効率性： システムリソースの可用性がリスクとして認識されている場合、特定の性能テストを実施することにより、リソースの利用状況（限られた RAM の割り当てなど）を調査することができる。

キャパシティ： システムに要求されるキャパシティの限界（ユーザー数やデータ量など）におけるシステムの振る舞いの問題をリスクとして認識した場合、システムアーキテクチャーの適合性を評価するために性能テストを実施することができる。

性能テストは、特定のシステムパラメーターの測定と分析を可能にする実験という形で行われることが多い。これらのテストは、システムの分析、設計、実装を支援するために反復的に実施され、アーキテクチャーの決定を可能にし、ステークホルダーの期待を形成するのに役立つ。

特に以下の性能テストの原則が関連している。

- さまざまなステークホルダーのグループ、特にユーザー、システム設計者、運用スタッフが定義した期待と合うテストケースでなければならない。
- テストケースは再現可能でなければならない。変更されていないシステムでテストケースを繰り返すことにより、統計的に同一の結果（指定された許容範囲内）が得られなければならない。
- テストケースからは、理解しやすく、ステークホルダーの期待値と容易に比較できるものが結果として現れなければならない。

- 完全または部分的なシステム、あるいは本番システムと同等のテスト環境のいずれかにおいて、リソースが許す限りテストケースを実施できる。
- プロジェクトで設定した期間内に、現実的なコストで実行可能なテストケースでなければならない。

[Molyneaux09]や[Microsoft07]の本は、性能テストの原理と実践的な側面についての確かな背景を提供している。

上記の 3 つの品質副特性はすべて、テスト対象システム（SUT）の拡張性に影響を与える。

1.2 性能テストのタイプ

性能テストにはさまざまなタイプがある。これらはそれぞれ、テストの目的に応じて、特定のプロジェクトに適用することができる。

- **性能テスト**

性能テストとは、システムやコンポーネントがさまざまな負荷にさらされたときの性能（応答性）に焦点を当てたあらゆるタイプのテストを含む包括的な用語である。

- **負荷テスト**

負荷テストでは、コントロールされた数の同時使用ユーザーまたはプロセスによって生成されるトランザクション要求に起因する、予想される現実的な負荷レベルの増加に対応するシステムの能力に焦点を当てる。

- **ストレステスト**

ストレステストでは、システムやコンポーネントが、想定された、あるいは仕様化したワークロードの限界に達した、あるいは限界を超えたピーク負荷を処理する能力に焦点を当てる。また、ストレステストは、アクセス可能な処理演算能力、利用可能な帯域幅、メモリーなどのリソースの可用性が低下した場合のシステムの処理能力を評価するためにも使用できる。

- **拡張性テスト**

拡張性テストでは、現在要求されている以上の将来的な効率性の要求を満たすためのシステムの能力に焦点を当てる。これらのテストの目的は、現在指定されている性能要件に違反したり故障したりすることなく、システムが成長する能力（ユーザー数の増加や保存データ量の増加など）を判断することである。拡張性の限界が明らかになれば、しきい値を設定し、本番環境でモニタリングすることで、問題が発生しそうなときに警告を発することができる。また、適切な量のハードウェアで本番環境を調整することもできる。

- **スパイクテスト**

スパイクテストでは、突然のピーク負荷に対してシステムが正しく応答し、その後、定常状態に戻る能力に焦点を当てる。

- **耐久テスト**

耐久テストでは、システムの運用状況に応じた時間軸でのシステムの安定性に焦点を当てる。このタイプのテストでは、最終的に性能が低下する、限界点での故障の原因となるといったリソース容量（例えば、メモリーリーク、データベース接続、スレッドプールなど）の問題がないことを検証する。

- **コンカレンシーテスト**

コンカレンシーテストでは、特定のアクションが同時に発生する状況（例：多数のユーザーが同時にログインする場合）の影響に焦点を当てる。コンカレンシーの問題は、発見し、再現することは非常に困難である。 特に、本番環境のようにテストがほとんど、あるいは全くコントロールできない環境で問題が発生した場合は困難になる。

- **キャパシティテスト**

キャパシティテストでは、あるシステムがどの程度のユーザー、そして／またはトランザクションをサポートしても規定の性能の目的を満たすことができるかを判断する。これらの目的は、トランザクションの結果として生じるデータ量に関連して述べられることもある。

1.3 性能テストにおけるテストの種類

性能テストの主なテストの種類には、静的テストと動的テストがある。

1.3.1 静的テスト

静的テストの活動は、機能適合性テストのためよりも性能テストのために重要になることが多い。これは、非常に多くの深刻な性能の欠陥がシステムのアーキテクチャーや設計で入り込むためである。これらの欠陥は、設計者やアーキテクトの誤解や知識不足によってもたらされる。また、これらの欠陥は、応答時間、スループット、リソース使用率などの目標値や、システムの予想される負荷や使用方法、制約条件などが要件に適切に盛り込まれていないために入り込むこともある。

性能に関する静的テストには、次のようなものがある。

- 性能面とリスクに焦点を当てた要件のレビュー
- データベーススキーマー、ER 図、メタデータ、ストアドプロシージャー、クエリーのレビュー
- システムおよびネットワークアーキテクチャーのレビュー
- システムコードの重要な部分（複雑なアルゴリズムなど）のレビュー

1.3.2 動的テスト

システムのビルド後、できるだけ早く動的な性能テストを開始すべきである。動的な性能テストの機会は以下の通りである。

- ユニットテストでは、プロファイリング情報を用いて潜在的なボトルネックを特定したり、および動的解析を用いてリソースの使用状況を評価したりする場合が該当する。
- コンポーネント統合テストでは、主要なユースケースやワークフローにわたって、特に異なるユースケースのフィーチャーを統合したり、ワークフローの「バックボーン」構造と統合したりする場合が該当する。
- システムテストでは、さまざまな負荷条件下にて全体的にエンドツーエンドの振る舞いを確認する場合が該当する。
- システム統合テストでは、特に重要なシステム間インターフェースを介したデータフローやワークフローが該当する。システム統合テストでは、「ユーザー」が他のシステムやマシンであることも珍しくない（例：センサー入力や他のシステムからの入力）。

- 受け入れテストでは、システムの適切な性能に対するユーザー、顧客、オペレーターの信頼を築き、実世界の条件下でシステムを微調整する。（ただし、通常はシステムの性能欠陥を見つけることはしない。）

システムテストやシステム統合テストのようなハイレベルのテストでは、正確な結果を得るために、現実的な環境、データ、負荷を使用することが重要である（第4章参照）。アジャイルやその他のイテレーティブ-インクリメンタルなライフサイクルにおいて、チームは、性能のリスクに対処するために最終的なイテレーションを待つのではなく、静的および動的な性能テストを初期のイテレーションに組み込む必要がある。

システムにカスタムハードウェアや新しいハードウェアが含まれている場合は、シミュレーターを使って初期の動的な性能テストを行うことができる。しかし、シミュレーターではリソースの制約や性能に関連する動作を十分に把握できないことが多いため、できるだけ早く実際のハードウェアでのテストを開始するのがよい。

1.4 負荷生成の概念

1.2 節で説明した性能テストタイプを実施するためには、代表的なシステム負荷をモデル化し、生成し、テスト対象システムに送信する必要がある。負荷は、機能テストケースに使用されるデータ入力に相当するが、主に以下の点で異なる。

- 性能テストの負荷は、1 つだけでなく、多くのユーザーの入力を表現する必要がある。
- 性能テストの負荷は、生成のために専用のハードウェアとツールを必要とする場合がある。
- 性能テストの負荷の生成をすることは、テスト実行に影響を与えることがある機能上の欠陥がテスト対象のシステムにないことが前提となる。

性能テストを行う際には、指定された負荷を効率的かつ確実に発生させることが重要な成功要因となる。負荷の生成にはさまざまな選択肢がある。

ユーザーインターフェースによる負荷生成

少数のユーザーによる利用を表せばよく、必要な入力を行うために必要な数のソフトウェアクライアントが利用可能である場合には、これが適切なアプローチだと言える。この方法は、機能テスト実行ツールと組み合わせて使用することもできるが、

シミュレーションするユーザーの数が増えると、すぐに実用的ではなくなってしまう。また、ユーザーインターフェース（UI）の安定性も重要な依存関係にある。頻繁な変更は、性能テストの再現性に影響を与え、メンテナンスコストにも大きな影響を与える可能性がある。エンドツーエンドのテストでは、UI を使ったテストが最も代表的なアプローチとなる。

多くの人を利用した負荷生成
このアプローチは、実際のユーザーを代表する多数のテスト担当者を確保できるかどうかにかかってくる。クラウドテスト（Crowd Testing）では、必要な負荷が発生するようにテスト担当者を編成する。これは、世界のどこからでもアクセス可能なアプリケーション（Web ベースなど）をテストするのに適した方法であり、ユーザーは幅広い種類の異なるデバイスや構成から負荷を発生させることができる。この方法では、非常に多くのユーザーを利用することができるが、生成される負荷は、他の方法ほど再現性と正確性に優れておらず、テスト担当者を編成するのがより複雑になる。

アプリケーションプログラミングインターフェース（API）による負荷生成
このアプローチは、データ入力に UI を使用するのと似ているが、UI の代わりにアプリケーションの API を使用して、テスト対象のシステムとユーザーのやりとりをシミュレーションする。そのため、UI の変更（遅延など）の影響を受けにくく、ユーザーが UI を通じて直接入力した場合と同じ方法でトランザクションを処理することができる。特定の API ルーチンを繰り返し呼び出す専用のスクリプトを作成することで、UI 入力を使用する場合に比べて、より多くのユーザーをシミュレーションすることができる。

キャプチャーした通信プロトコルによる負荷生成
このアプローチでは、テスト対象のシステムとユーザーのやりとりを通信プロトコルレベルでキャプチャーし、これらのスクリプトを再生することで、非常に多くのユーザーを再現性と信頼性の高い方法でシミュレーションする。このツールを使用したアプローチについては、4.2.6 および 4.2.7 項で説明する。

1.5 一般的な性能効率性の故障モードとその原因

動的テストで発見される性能の故障モードは確かにさまざまだが、以下に一般的な故障（システムクラッシュを含む）の例と、その典型的な原因を示す。

すべての負荷レベルで応答時間が遅い
場合によっては、負荷に関係なく、応答時間が受け入れられないことがある。これは、データベースの設計や実装の不備、ネットワークの遅延、その他のバックグラウンドの負荷など、根本的な性能の問題が原因となっている可能性がある。このような問題は、性能テストだけでなく、機能テストや使用性テストでも確認できるので、テストアナリストはこれらの問題に注意を払い、報告する必要がある。

中程度から高程度の負荷時に応答時間が遅い
場合によっては、中程度から高程度の負荷がかかると、その負荷が通常の想定される許容範囲内であっても、応答時間が許容できないほど悪化することがある。このような問題の背景には、1 つまたは複数のリソースが飽和していることや、バックグラウンドの負荷が変化していることなどがある。

時間の経過とともに応答時間が悪化する
場合によっては、時間の経過とともに応答時間が徐々にもしくは顕著に悪化したり、深刻な状態になったりすることがある。その原因としては、メモリーリーク、ディスクの断片化、ネットワーク負荷の長期的な増大、ファイルリポジトリの増大、データベースの予期せぬ増大などが挙げられる。

高負荷、または限界を超えたときの不適切、またはルールに沿わないエラー処理
場合によっては、応答時間は許容範囲内でも、高負荷や限界を超えた負荷レベルではエラー処理が劣化する。その原因としては、リソースプールの不足、キューやスタックのサイズ不足、タイムアウトの設定が短すぎることなどが挙げられる。

上記の一般的な故障の具体例としては、以下のようなものがある。

- 企業のサービスに関する情報を提供する Web ベースのアプリケーションが、ユーザーのリクエストに対して 7 秒以内（業界の一般的な経験則）に応答しない。特定の負荷条件では、システムの性能効率性が達成できない。
- 突然の大量のユーザーリクエスト（例：大規模なスポーツイベントのチケット販売）があった場合に、システムがクラッシュしたり、ユーザーの入力に対応したりできない。このような多数のユーザーを処理するためのシステムのキャパシティが不十分である。
- ユーザーが大量のデータをリクエストすると、システムの応答時間が著しく低下する（例：サイズが大きく重要なレポートをダウンロードできるように Web サイトへ掲載している）。発生したデータ量を処理するシステムのキャパシティが不足している。

- オンライン処理開始が必要な時間までにバッチ処理が完了しない。バッチ処理に許された時間帯に対してバッチ処理の実行時間が長すぎる。
- リアルタイムシステムの RAM が不足するのは、並列プロセスが動的メモリーを大量に要求し、それを時間内に解放できない場合である。RAM 容量が適切でないか、RAM への要求に適切な優先順位が付けられていない。
- リアルタイムシステムコンポーネント B に入力を供給するリアルタイムシステムコンポーネント A は、必要な速度で更新を計算することができない。システム全体が時間内に応答できず、故障する可能性がある。コンポーネント A のコードモジュールを評価し、必要な更新レートを達成できるように修正しなければならない（「性能プロファイリング」）。

2. 性能測定の基礎 – 55 分

キーワード

測定、メトリクス

「性能測定の基礎」の学習の目的

2.1 性能テストで収集する典型的なメトリクス
PTFL-2.1.1 (K2) 性能テストで収集する典型的なメトリクスを理解する。

2.2 性能テストの結果の集約
PTFL-2.2.1 (K2) 性能テストの結果の集約が必要な理由を説明できる。

2.3 性能メトリクスの主な情報源
PTFL-2.3.1 (K2) 性能メトリクスの主要な情報源を理解する。

2.4 性能テストの典型的な結果
PTFL-2.4.1 (K1) 性能テストの典型的な結果を想起する。

2.1 性能テストで収集する典型的なメトリクス

2.1.1 なぜ性能メトリクスが必要なのか

性能テストのゴールを定め、性能テストの結果を評価するためには、正確な測定と、その測定値から得られるメトリクスが不可欠である。性能テストは、どのような測定値やメトリクスが必要なのかを最初に理解せずに実施してはいけない。このアドバイスを無視すると、以下のようなプロジェクトリスクになる。

- 運用の目的を満たすために、性能レベルが許容できるかが不明
- 性能要件が測定可能な表現で定義されていない
- 性能の低下を予測するような傾向を特定できない可能性
- 性能テストの実際の結果が、許容できる性能、および／または許容できない性能を定義した一連のベースラインとなる性能尺度と比較して評価できない
- 性能テストの結果を 1 人または複数人の主観的な意見に基づいて評価する
- 性能テストツールが提供する結果が理解できない

2.1.2 性能測定値とメトリクスの収集

正確な方法でメトリクスを取得して表現することは、他の形式の測定でもそうであるのと同様に可能である。したがって、この節で説明するメトリクスと測定値のいずれも、特定の文脈で意味を持つように定義することができ、またそうすべきである。そのためには、初期にテストケースを実行した結果から、どのメトリクスをさらに改良し、どのメトリクスを追加する必要があるかを学ぶことが重要である。

例えば、応答時間のメトリクスは、一連の性能メトリクスのどこかに含められると考えられる。しかし、意味のある実用的なメトリクスとするためには、時間帯、同時使用ユーザーの数、処理されるデータ量などの観点から、応答時間のメトリクスをさらに定義する必要がある。

特定の性能テストで収集されるメトリクスは、以下の条件によって異なる。

- ビジネスの状況（ビジネスプロセス、顧客やユーザーの行動、ステークホルダーの期待など）
- 運用の状況（技術とその使われ方）
- テストの目的

例えば、国際的な電子商取引サイトの性能テストで選択されるメトリクスと、医療機器の機能をコントロールするための組込みシステムの性能テストで選択されるメトリクスは異なる。

性能の測定値やメトリクスを分類する一般的な方法は、性能の評価が必要とされる技術環境、ビジネス環境、運用環境を考慮することである。

以下に示す測定値やメトリクスのカテゴリーは、性能テストから一般的に得られるものである。

技術環境

性能メトリクスは、以下の一覧に示すように、技術環境の種類によって異なる。

- Web ベース
- モバイル
- モノのインターネット（IoT）
- デスクトップクライアントデバイス
- サーバーサイドの処理
- メインフレーム

- データベース
- ネットワーク
- その環境で稼働しているソフトウェアの性質（例：組込み）

メトリクスには以下のようなものがある。

- 応答時間　（例：1 トランザクションあたり、1 同時使用ユーザーあたり、ページ読み込み時間）
- 資源効率性（例：CPU、メモリー、ネットワーク帯域、ネットワークレイテンシー、使用可能なディスクスペース、I/O レート、アイドルおよびビジー状態のスレッド）
- 主要なトランザクションのスループット率（例：一定期間に処理できるトランザクション数）
- バッチ処理時間（例：待ち時間、スループット時間、データベースの応答時間、完了時間）
- 性能に影響を与えるエラーの数
- 完了時間（例：データの作成、読み込み、更新、削除にかかる時間）
- 共有リソースのバックグラウンド負荷（特に仮想化環境での負荷）
- ソフトウェアメトリクス（例：コードの複雑性）

ビジネス環境

ビジネスまたは機能的な観点から、性能メトリクスには以下のようなものがある。

- ビジネスプロセスの効率性（例：通常、代替、例外ユースケースのフローを含むビジネスプロセス全体の実行速度）
- データ、トランザクション、および他の実行された作業単位のスループット（例：1 時間あたりに処理された注文、1 分あたりに追加されたデータ行）
- サービスレベルアグリーメント（SLA）の遵守率または違反率（例：単位時間あたりの SLA 違反率）
- 使用範囲（例：ある時点でタスクを実行している全世界または各国のユーザーの割合）
- 並列使用（例：タスクを同時に実行しているユーザーの数）
- 利用のタイミング（例：負荷の高い時間帯に処理された注文数）

運用環境

性能テストの運用面では、本来はユーザーの目には触れないと考えられているタスクに焦点を当てる。これには次のようなものがある。

- 運用プロセス（例：環境の立ち上げ、バックアップ、シャットダウン、再開などに要する時間）
- システムの復元（例：バックアップからのデータ復元に要する時間）
- アラートと警告（例：システムがアラートや警告を発するのに必要な時間）

2.1.3 性能メトリクスの選択

必要以上に多くのメトリクスを集めることは必ずしもよいことではないことに留意すべきである。選択された各メトリクスには、一貫した収集と報告の手段が求められる。性能テストの目的に役立つ、入手可能な一連のメトリクスを定義することが重要である。

例えば、GQM（Goal-Question-Metric）アプローチは、性能ゴールとメトリクスを一致させるのに役立つ方法である。これは、まずゴールを設定し、次にそのゴールが達成されたかを知るために質問をする。メトリクスは、質問への回答が測定可能であることを保証するために、各質問に関連付けられている。(GQM アプローチのより完全な説明については、エキスパートレベルシラバス「テストプロセスの改善」[ISTQB_ELTM_ITP_SYL]の 4.3 節を参照のこと)。GQM アプローチは、必ずしも性能テストプロセスに常に適合しないことに留意すべきである。例えば、いくつかのメトリクスはシステムの健全性を表しており、ゴールとは直接リンクしていない。

重要なことは、最初の測定値を定義して取得した後、真の性能レベルを理解し、是正措置が必要な箇所を特定するために、更なる測定値やメトリクスが必要になる場合があることを認識することである。

2.2 性能テストの結果を集約する

性能メトリクスを集約する目的は、システム性能の全体像を正確に把握し、表現できるようにすることである。性能メトリクスを詳細なレベルでしか見ていないと、正しい結論を導き出すことは、特にビジネスステークホルダーにとっては難しいかもしれない。

多くのステークホルダーにとっての主な関心ごとは、システム、Web サイト、その他のテスト対象物の応答時間が許容範囲内かである。

性能メトリクスのより深い理解が得られれば、そのメトリクスを集約して以下のことができる。

- ビジネスやプロジェクトのステークホルダーが、システム性能の「全体像」の状況を把握できる。
- 性能の傾向を把握できる。
- 性能メトリクスを分かりやすくレポートできる。

2.3 性能メトリクスの主な情報源

メトリクス収集のための付加によるシステム性能への影響（「プローブ効果」と呼ばれる）は最小限にとどめなければならない。さらに、性能メトリクスを収集する必要がある量、正確さ、スピードを考慮すると、ツールの使用が必須となる。ツールの併用は珍しいことではないが、テストツールの使用に冗長性が生じたり、その他の問題が発生したりする可能性がある（4.4 節参照）。
性能メトリクスには 3 つの重要なソースがある。

性能テストツール
すべての性能テストツールは、テストの結果として測定値やメトリクスを提供する。ツールによって、表示されるメトリクスの数、測定値の表示方法、ユーザーが特定の状況に合わせて測定値をカスタマイズできる機能などが異なる（5.1 節も参照）。

一部のツールはテキスト形式で性能メトリクスを収集、表示し、より強力なツールはダッシュボード形式でグラフィカルに性能メトリクスを収集、表示する。多くのツールは、テストの評価やレポート作成を容易にするために、測定値をエクスポートする機能を備えている。

性能モニタリングツール
性能モニタリングツールは、性能テストツールのレポート機能を補完するために採用されることが多い（5.1 節も参照）。さらに、モニタリングツールは、システムの性能を継続的にモニタリングし、性能の低下、システムエラーやアラートの増加をシステム管理者に警告するためにも使用される。また、これらのツールは、不審な動作（サービス拒否攻撃や分散型サービス拒否攻撃など）を検出して通知するためにも使用できる。

ログ分析ツール
サーバーのログをスキャンし、そこからメトリクスを算出する。これらのツールの中には、データをグラフィカルに表示するチャートを作成できるものもある。

通常、エラー、アラート、警告はサーバーログに記録される。これらには以下のようなものがある。

- CPU の使用率が高いといった高いリソースの利用、大量のディスクストレージの消費、帯域幅の不足
- メモリーの消耗など、メモリーに関するエラーや警告
- 特にデータベース操作時に起こる、デッドロックやマルチスレッドの問題
- SQL 例外や SQL タイムアウトなどのデータベースエラー

2.4 性能テストの典型的な結果

機能テストでは、特に、仕様化した機能要件やユーザーストーリーの機能要素を検証する場合、通常、期待結果を明確に定義し、テスト結果を解釈してテストの合否を判断する。例えば、月次の売上報告書で正しい合計値と間違った合計値を示す場合が該当する。

機能適合性を検証するテストケースでは、明確に定義されたテストオラクルの恩恵を受けることが多いが、性能テストではこのような情報源がないことがよくある。ステークホルダーが性能要件を明確にすることが苦手なだけでなく、ビジネスアナリストやプロダクトオーナーの多くも、そのような要件を引き出すことが苦手である。テスト担当者が入手するのは、期待されるテスト結果を定義するための限定的なガイドとなってしまうことが多い。

性能テストの結果を評価する際には、その結果を詳しく見ることが重要である。最初に得られる加工していない結果は、一見明らかに良好な全体結果の下に性能に関する故障が隠れていることがあり、誤解を招く可能性がある。例えば、潜在的なボトルネックとなるすべての主要なリソースの使用率が 75%以下で良好にもかかわらず、主要なトランザクションやユースケースのスループットや応答時間が桁違いに遅い場合がある。

具体的な評価結果は、実行するテストによって異なるが、多くの場合、2.1 節で述べたものが含まれる。

3. ソフトウェアライフサイクルにおける性能テスト – 195 分

キーワード

メトリクス、リスク、ソフトウェア開発ライフサイクル、テスト結果記録

「ソフトウェアライフサイクルにおける性能テスト」の学習の目的

3.1 主な性能テストの活動

PTFL-3.1.1 (K2) 性能テストの主な活動を理解する。

3.2 アーキテクチャーの違いによる性能リスク

PTFL-3.2.1 (K2) 異なるアーキテクチャーの性能リスクの典型的なカテゴリーを説明できる。

3.3 ソフトウェア開発ライフサイクルにおける性能リスク

PTFL-3.3.1 (K4) ソフトウェア開発ライフサイクルにおいて、ある製品の性能リスクを分析することができる。

3.4 性能テストの活動

PTFL-3.4.1 (K4) 与えられたプロジェクトを分析し、ソフトウェア開発ライフサイクルの各フェーズに適切な性能テスト活動を決定することができる。

3.1 性能テストの主要な活動

性能テストは反復的に行われる。各テストでは、アプリケーションやシステムの性能に関する貴重な洞察が得られる。1 回のテストで収集された情報は、アプリケーションやシステムのパラメーターを修正または最適化するために使用する。そして、次のテストの反復では、修正した結果が示され、テストの目的が達成されるまで、これを繰り返す。

性能テストの活動は、ISTQB®のテストプロセス[ISTQB_FL_SYL]に沿って行われる。

テスト計画

テスト計画は性能テストでは特に重要である。なぜなら、テスト環境、テストデータ、ツール、人材などを割り当てる必要があるためである。また、テスト計画は性能テストのスコープを設定する活動でもある。

テスト計画中に、リスクの識別とリスク分析の活動が完了し、テスト計画の文書（テスト計画書、レベルテスト計画書など）の中で関連情報を更新する。テスト計画が必要に応じて見直され、修正されるのと同様に、リスクの状況の変化を反映して、リスク、リスクレベル、リスクステータスを修正する。

テストのモニタリングとコントロール
以下のような性能効率性に影響を与える可能性のある問題が発生した場合のアクション計画を提供するために、コントロールの手段を定義する。

- 性能テストのテストケースのために計画した望ましい負荷をインフラが生成できない場合に、負荷生成のキャパシティを増加
- ハードウェアの変更、刷新、入替
- ネットワークコンポーネントの変更
- ソフトウェア実装の変更

終了基準の達成度合いを確認するために性能テストの目的を評価する。

テスト分析
効果的な性能テストは、性能要件、テスト目的、サービスレベルアグリーメント（SLA）、IT アーキテクチャー、プロセスモデル、およびテストベースを構成するその他のアイテムの分析に基づく。この活動は、スプレッドシートやキャパシティプランニングツールを使用した、システムリソースの要件や動作のモデル化と分析が役立つ場合がある。

負荷レベル、タイミング条件、テスト対象となるトランザクションなどの具体的なテスト条件を識別する。そして、必要な性能テストタイプ（例えば、負荷、ストレス、拡張性）を決定する。

テスト設計
性能テストケースを設計する。これらは通常、より大規模で複雑な性能テストの構成要素として使用できるように、モジュール形式で作成する（4.2 節参照）。

テスト実装
実装段階では、性能テストケースは、性能テスト手順に組み込まれる。これらの性能テスト手順は、ユーザーが通常行うステップや、性能テスト中にカバーされるべき他の機能的なアクティビティを反映したものでなければならない。

テスト実装の活動は、各テスト実行の前にテスト環境を確立したり、再設定したりすることである。性能テストはデータ駆動が典型的であるため、本番での使われ方をシミュレーションできるように、実際の本番データの量や種類を再現するテストデータを確立するプロセスが必要である。

テスト実行
テスト実行は、性能テストを実施する際に行われ、多くの場合は性能テストツールを使用する。テスト結果を評価し、システムの性能が要件やその他の目的を満たしているかどうかを判断する。欠陥はすべてレポートする。

テスト完了
性能テストの結果は、テストサマリーレポートとしてステークホルダー（アーキテクト、マネージャー、プロダクトオーナーなど）へ提供する。テスト結果は、テスト結果の意味をシンプルにするため、集約したメトリクスで表現することが多い。テキストベースのメトリクスよりも理解しやすいダッシュボードなどの視覚的なレポート手段が性能テストの結果を表現するためによく使用される。

性能テストは、すべてのテストレベル（コンポーネントテスト、統合テスト、システムテスト、システム統合テスト、受け入れテスト）で何度も実施されるという点で、継続的な活動であると考えられる。設計したテストケースがテスト完了に到達できて、性能テストの期間が終了したときに、テストツール資産（テストケースやテスト手順）、テストデータ、およびその他のテストウェアを保存して、のちに行うシステムメンテナンス活動で利用するために他のテスト担当者に引き継ぐ。

3.2 アーキテクチャーの違いによる性能リスクの分類

前述の通り、アプリケーションやシステムの性能は、アーキテクチャー、アプリケーション、ホスト環境によって大きく異なる。すべてのシステムの性能リスクの完全なリストを提供することは不可能であるが、以下のリストには、特定のアーキテクチャーに関連するいくつかの典型的なタイプのリスクが含まれている。

シングルコンピューターシステム
仮想化されていない 1 台のコンピューター上で完全に動作するシステムやアプリケーションのことである。以下の理由で性能が低下することがある。

- 過剰なリソース消費。例えば、メモリーリーク、セキュリティソフトウェアなどのバックグラウンドアクティビティ、低速のストレージサブシステム（例：低速の外

部デバイス、またはディスクの断片化）、オペレーティングシステムの誤った管理などが該当する。
- アルゴリズムの非効率な実装による、利用可能なリソース（メインメモリーなど）を活用できないことと、その結果による必要以上の実行速度低下。

多層システム
データベースサーバー、アプリケーションサーバー、プレゼンテーションサーバーなど、それぞれが特定のタスクを実行する複数のサーバー上で動作するシステムオブシステムズのことである。各サーバーは当然ながら1つのコンピューターであり、先に挙げたようなリスクがある。また、貧弱で拡張性のないデータベース設計、ネットワークのボトルネック、単一サーバーの帯域や容量の不足などにより、性能が低下する可能性がある。

分散システム
多層アーキテクチャーに類似しているが、例えば、注文者の地理的な位置に応じて異なる在庫データベースにアクセスする電子商取引システムのように、さまざまなサーバーが動的に変化する場合があるシステムオブシステムズのことである。このアーキテクチャーでは、多層アーキテクチャーに関連するリスクに加えて、信頼性の低い、あるいは予測不可能なリモートサーバーとの間で、あるいはリモートサーバーを経由して、重要なワークフローやデータフローを処理することにより、性能に問題が生じる可能性がある。特に、このようなリモートサーバーが定期的な接続障害や断続的な激しい高負荷状態に陥った場合には、この問題が発生する。

仮想化システム
物理的なハードウェアが複数の仮想コンピューターをホストするシステムのことである。これらの仮想マシンは、単一のコンピューターシステムやアプリケーションをホストする場合もあれば、多層アーキテクチャーや分散アーキテクチャーのサーバーの一部をホストする場合もある。仮想化特有の性能リスクとしては、すべての仮想マシンにわたりハードウェアに過剰な負荷がかかることや、ホスト仮想マシンの設定が不適切でリソースが不足することなどが挙げられる。

ダイナミック／クラウドベースシステム
負荷の増加に応じてキャパシティを増加させ、要求に応じて拡張できる機能を備えているシステムのことである。これらは典型的には分散システムや仮想化した多層システムであるが、これらのアーキテクチャーに関連する性能リスクを特に軽減するために設計された自己スケーリング機能を備えている。しかし、初期設定時やそ

の後のアップデート時に、これらの機能の適切な設定に失敗することについてのリスクが存在する。

クライアントサーバーシステム
クライアント上で動作し、ユーザーインターフェースを介して単一のサーバー、多層サーバー、または分散サーバーと通信するシステムのことである。クライアント上でコードが実行されるため、シングルコンピューターのリスクがそのコードにおいて該当し、前述のサーバー側の問題も同様に該当する。さらに、接続速度や信頼性の問題、クライアントの接続ポイント（公共の Wi-Fi など）でのネットワークの混雑、ファイアウォールやパケットインスペクション、サーバーの負荷分散などによる潜在的な問題などにより、性能上のリスクが存在する。

モバイルアプリケーション
スマートフォンやタブレットなどのモバイルデバイス上で動作するアプリケーションである。このようなアプリケーションには、クライアントサーバーやブラウザー（Web アプリケーション）をベースにしたアプリケーションで述べたようなリスクがある。加えて、モバイル機器で利用可能なリソースや接続性が限られており、かつ変動するため、性能上の問題が発生する可能性がある（場所、バッテリー残量、充電状態、機器上の利用可能なメモリー、温度などに影響される）。また、加速度計や Bluetooth などのデバイスのセンサーや無線を使用するアプリケーションでは、これらのソースからの遅いデータフローが問題となる可能性がある。最後に、モバイルアプリケーションは、他のローカルなモバイルアプリケーションやリモートの Web サービスと頻繁にやりとりすることが多く、これらが性能効率のボトルネックになる可能性がある。

組込みリアルタイムシステム
自動車（エンターテインメントシステムやインテリジェントブレーキシステムなど）、エレベーター、交通信号、暖房・換気・空調（HVAC）システムなどの日常生活の中で動作し、さらにはそれらをコントロールするシステムである。これらのシステムは、インターネットに接続されているため、（ますます増加する）接続性に関する問題も含め、モバイル機器としての多くのリスクを抱えている。通常、モバイルゲームの性能の低下はユーザーにとって安全上の問題ではないが、自動車のブレーキシステムはこのような処理速度の低下が致命的な問題となる可能性がある。

メインフレームアプリケーション
多くの場合、数十年前に開発されており、データセンター内で、時にはバッチ処理によってミッションクリティカルなビジネス機能をサポートしているアプリケーションのことである。ほとんどの場合、当初の設計通りでは、予測可能性が非常に高

く動作が早い。しかし、これらのアプリケーションの多くは、現在、API や Web サービス、またはデータベースを介してアクセスされており、その結果、予期せぬ負荷がかかり、既存のアプリケーションのスループットに影響を与えることがある。

あらゆるアプリケーションやシステムには、上記のアーキテクチャーが 2 つ以上組み込まれている可能性があり、その場合、関連するすべてのリスクがそのアプリケーションやシステムに該当することになる。実際、極端なレベルの相互作用と接続がルールとなっているモノのインターネット（IoT）とモバイルアプリケーションの爆発的な普及を考えると、この 2 つにはすべてのアーキテクチャーが何らかの形でアプリケーションに存在する可能性があり、すべてのリスクが該当する可能性がある。

アーキテクチャーは性能リスクに大きな影響を与える重要な技術的決定であることは明らかだが、その他の技術的決定もリスクに影響を与え、リスクを生み出す。例えば、C や C++のようにヒープメモリーを直接管理できる言語ではメモリーリークが多く発生するし、リレーショナルデータベースと非リレーショナルデータベースでは性能の問題が異なる。このような判断は、個々の関数やメソッドの設計にまで及ぶ（例えば、反復アルゴリズムではなく再帰アルゴリズムを選択するなど）。テスト担当者がこのような決定について知ることができるか、あるいは影響を与えることができるかは、組織やソフトウェア開発ライフサイクルにおけるテスト担当者の役割や責任によって異なる。

3.3 ソフトウェア開発ライフサイクルにおける性能リスク

ソフトウェアプロダクトの品質に対するリスクを分析するプロセス全般については、他の ISTQB® のシラバスで議論されている（例えば、[ISTQB_FL_SYL] や [ISTQB_ALTM_SYL]を参照）。また、特定の品質特性に関連した具体的なリスクや考慮事項についての議論（例えば、[ISTQB_UT_SYL]）や、ビジネスや技術的な観点からの議論（例えば、[ISTQB_ALTA_SYL]や[ISTQB_ALTTA_SYL]をそれぞれ参照）も見られる。この節では、プロセス、参加者、考慮事項が変化する方法を含め、プロダクト品質に対する性能関連リスクに焦点を当てている。

プロダクトの品質に対する性能関連リスクのリスク分析プロセスは以下の通り。

1. 時間効率性、資源効率性、キャパシティなどの特性に着目し、プロダクト品質に対するリスクを識別する。

2. 識別したリスクを評価し、関連するアーキテクチャーの分類（3.2 節参照）に対応していることを確認する。 識別した各リスクについて、明確に定義された基準を用いて、可能性と影響の観点から総合的なリスクレベルを評価する。
3. リスクアイテムの性質とリスクのレベルに基づいて、各リスクアイテムに対して適切なリスク軽減措置をとる。
4. リリース前にリスクが十分に軽減されていることを確実にするために、継続的にリスクに対処する。

一般的な品質リスク分析と同様に、このプロセスの参加者にはビジネスと技術の両方のステークホルダーが含まれるべきである。性能関連のリスク分析の場合、ビジネス上のステークホルダーには、本番での性能問題が顧客、ユーザー、事業、その他の下流のステークホルダーに対して実際にどのような影響を与えるかについて知っている人たちが含まれなければならない。ビジネス上のステークホルダーは、ビジネスの観点からさまざまな要因がリスクに作用し、リスクを生み出し、故障の影響度合いを左右することについて理解しなければならない。
さまざまな要因には、意図した使用方法、ビジネス上や社会上または安全上の重要性、潜在的な金銭的および／または評判上の損害、民事上または刑事上の法的責任、およびこれらと同様の複数の要因がある。

さらに、技術的なステークホルダーは、技術的な観点からさまざまな要因が性能リスクに作用し、リスクを生み出し、欠陥の可能性の度合いを左右することについて理解しなければならない。さまざまな要因には、アーキテクチャー、設計、および実装の決定がある。

選ばれた特定のリスク分析プロセスは、適切なレベルの形式と厳密さを持つべきである。性能関連のリスクについては、リスク分析プロセスを早期に開始し、定期的に繰り返すことが特に重要である。言い換えれば、テスト担当者は、システムテストレベルやシステム統合テストレベルの終盤に行われる性能テストに全面的に依存することを避けるべきである。多くのプロジェクト、特に大規模で複雑なシステムオブシステムズのプロジェクトでは、プロジェクトの初期に行われた要件、設計、アーキテクチャー、実装の判断に起因する性能欠陥の発見が遅れたために、不運にも想定外の事象に見舞われている。そのため、ソフトウェア開発のライフサイクルを通じて、性能リスクの識別、アセスメント、軽減措置、マネジメントを反復的に行うことに重点を置くべきである。

例えば、大量のデータをリレーショナルデータベースで扱う場合、貧弱なデータベース設計に起因した多対多の結合の性能低下は、システムテストのような大規模な

テストデータを用いた動的テストで初めて明らかになることがある。しかし、経験豊富なデータベースエンジニアを巻き込んだ慎重なテクニカルレビューによって、データベース実装前に問題点を予測することができる。反復的なアプローチでは、このようなレビューの後でリスクを識別し、再度評価を行うことができる。

さらに、リスクの軽減とマネジメントは、動的テストだけでなく、ソフトウェア開発プロセス全体に影響を与えなければならない。例えば、予想されるトランザクション数や同時ユーザー数など、性能に関わる重要な決定事項をプロジェクトの初期段階で特定できない場合、設計やアーキテクチャーの決定において、高度に可変な拡張性（要求に応じて対応できるクラウドベースのコンピューティングリソースなど）を考慮することが重要になる。これにより、早期にリスクを軽減するための意思決定を行うことができる。

優れた性能エンジニアリングによって、プロジェクトチームは、システム統合テストやユーザー受け入れテストなど、よりハイレベルのテストで重大な性能欠陥が遅れて発見されることを避けることができる。プロジェクトの遅い段階で発見された性能の欠陥は、非常に大きなコストとなり、プロジェクト全体の中止につながることもある。

他の種類の品質リスクと同様に、性能関連のリスクは完全に回避することはできない。すなわち、性能に関する本番環境での故障のリスクは常に存在する。したがって、リスクマネジメントのプロセスには、そのプロセスに関わるビジネスおよび技術のステークホルダーに対して、残存するリスクのレベルについて現実的かつ具体的な評価を提供することが含まれなければならない。例えば、単に「はい、お客様のチェックアウト時に長時間の遅延が発生する可能性はまだあります。」と言うだけでは、どの程度のリスク軽減が行われたのか、あるいはどの程度のリスクが残っているのかが分からないため、役に立たない。その代わり、一定のしきい値以上の遅延が発生する可能性のあるお客様の割合を明確に示すことで、状況を理解してもらうことができる。

3.4 性能テストの活動

性能テストの活動は、ソフトウェア開発ライフサイクルの種類によって、編成が異なったり、実施方法が異なったりする。

シーケンシャル開発モデル

シーケンシャル開発モデルにおける性能テストの理想的な実践事例は、プロジェクトの開始時に定義される受け入れ基準の一部として性能基準を含めることである。テストのライフサイクルの視点を強化するために、性能テスト活動は、ソフトウェア開発のライフサイクルを通して実施されるべきである。プロジェクトの進行に伴い、各性能テスト活動は、以下に示すように、それ以前の活動で定義された項目に基づいて行われるべきである。

- 企画段階 - システムの性能ゴールがプロジェクトの受け入れ基準として定義されていることを確認する。
- 要件- 性能要件が定義され、ステークホルダーのニーズを正しく表していることを確認する。
- 分析と設計- システム設計が性能要件を反映していることを検証する。
- コーディング/実装 - コードが効率的であり、性能の面で要件や設計を反映していることを検証する。
- コンポーネントテスト- コンポーネントレベルの性能テストを実施する。
- コンポーネント統合テスト- コンポーネント統合レベルでの性能テストを実施する。
- システムテスト- 本番環境を再現するハードウェア、ソフトウェア、手順、データを含むシステムレベルでの性能テストを実施する。システムインターフェースは、性能を忠実に再現することを条件にシミュレーションすることができる。
- システム統合テスト-本番環境と同等のシステム全体での性能テストを行う。
- 受け入れテスト- システムの性能が、当初述べられたユーザーニーズと受け入れ基準を満たしているかどうかを検証する。

イテレーティブ、およびインクリメンタル開発モデル

アジャイルなどの開発モデルでは、性能テストもイテレーティブ、およびインクリメンタルな活動とみなされる（[ISTQB_FL_AT]参照）。性能テストは、最初のイテレーションの一部として、あるいは性能テストに完全に特化したイテレーションとして行われる。しかし、これらのライフサイクルモデルでは、性能テストの実行は、性能テストを担当する別のチームが行うことができる。

継続的インテグレーション（CI）は、イテレーティブ、およびインクリメンタルなソフトウェア開発ライフサイクルにおいて一般的に行われており、テストの高度な自動実行を可能にする。CI におけるテストの最も一般的な目的は、リグレッションテストを行い、各ビルドが安定していることを確認することである。ビルドレベルで実行されるようにテストが設計されていれば、性能テストも CI で実行される自動

テストの一部になる。しかし、機能のテストケースの自動化とは異なり、以下のような考慮事項がある。

- 性能テスト環境のセットアップ - これには多くの場合、クラウドベースの性能テスト環境など、要求に応じて利用可能なテスト環境が必要である。
- CI で自動化する性能テストケース決定- CI のテストケースのためにとれる時間が短いため、CI の性能テストケースは、イテレーション中の他の時期に専門家チームが実施する、より広範な性能テストケースのサブセットになることがある。
- CI のための性能テストケース作成- CI の一環としての性能テストの主な目的は、変更が性能に悪影響を与えないことを確認することである。ビルドの変更内容によっては、新たな性能テストが必要になる場合がある。
- アプリケーションやシステムの一部を対象とした性能テストケースの実行- これには、適用可能なテストケースのサブセットを選択でき、迅速な性能テストが可能なツールやテスト環境が必要となることが多い。

イテレーティブ、およびインクリメンタルなソフトウェア開発のライフサイクルで行われる性能テストにもライフサイクル活動がある。

- リリース計画 - この活動では、最初のイテレーションから最終のイテレーションまで、1 回のリリースの中のすべてのイテレーションの観点で性能テストを検討する。性能のリスクを識別し、評価し、軽減策を計画する。これには、アプリケーションをリリースする前の最終的な性能テストの計画も含まれることが多い。
- イテレーション計画- 各イテレーションの文脈では、イテレーション内および各イテレーションの完了時に性能テストを実施することができる。性能リスクは、各ユーザーストーリーについてより詳細に評価される。
- ユーザーストーリーの作成- ユーザーストーリーは、アジャイルの方法論における性能要件の基礎となることが多く、具体的な性能基準は関連する受け入れ基準に記述される。これらは「非機能的」ユーザーストーリーと呼ばれている。
- 性能テストケースの設計- 特定のユーザーストーリーに記載されている性能要件と基準をテストケースの設計に使用する（4.2 節参照）。
- コーディング／実装 - コーディングの際には、コンポーネントレベルでの性能テストを行うことがある。性能効率性を最適化するためのアルゴリズムのチューニングが一例として挙げられる。
- テスト／評価 - テストの実施は、通常は開発活動に密接して行われるが、イテレーションにおける性能テストの範囲と目的によっては、性能テストを別の活動として実施することもある。例えば、性能テストの目的が、完成した複数のユーザーストーリーのセットとしてのイテレーションにて性能をテストすることである場合、単一

のユーザーストーリーの性能テストで行われるものよりも広い範囲の性能テストが必要になる。これは、性能テスト専用のイテレーションでスケジュールされることがある。

- リリース- リリースによってアプリケーションが本番環境に導入されるため、実際の使用時にアプリケーションが求められるレベルの性能を達成しているかどうかを判断するために、性能をモニタリングする必要がある。

市販ソフトウェア（COTS）およびその他の供給者/取得者モデル

多くの組織では、アプリケーションやシステムを自ら開発するのではなく、ベンダーからの供給やオープンソースプロジェクトからソフトウェアを入手する立場にある。このような供給者/取得者モデルでは、性能は重要な検討事項であり、供給者（ベンダー/開発者）と取得者（顧客）の両方の視点からのテストが必要である。

アプリケーションの提供元に関わらず、性能が要件を満たしているか妥当性確認をするのは、多くの場合、顧客の責任となる。カスタマイズされたベンダー開発ソフトウェアの場合には、ベンダーと顧客の間の契約の一部として、性能要件と関連する受け入れ基準を規定する必要がある。COTS アプリケーションの場合は、顧客が単独で、現実的なテスト環境でプロダクトの性能をデプロイ前にテストする責任がある。

4. 性能テストのタスク– 475 分

キーワード

コンカレンシー、ロードプロファイル、負荷生成、運用プロファイル、ランプダウン、ランプアップ、システムオブシステムズ、システムスループット、テスト計画書、シンクタイム、仮想ユーザー

「性能テストのタスク」の学習の目的

4.1 計画

PTFL-4.1.1 (K4) 関連する情報から、性能テストの目的を導き出す。
PTFL-4.1.2 (K4) プロジェクトの性能の目的を考慮した性能テスト計画書の概要を説明する。
PTFL-4.1.3 (K4) 計画した性能テストの根拠をさまざまなステークホルダーが理解できるようなプレゼンテーションを作成する。

4.2 分析・設計・実装

PTFL-4.2.1 (K2) 性能テストで使用される典型的なプロトコルの例を挙げる。
PTFL-4.2.2 (K2) 性能テストにおけるトランザクションの概念を理解する。
PTFL-4.2.3 (K4) システム利用に関する運用プロファイルを分析する。
PTFL-4.2.4 (K4) 性能の目的に対して、運用プロファイルからロードプロファイルを作成する。
PTFL-4.2.5 (K4) 性能テストの開発において、スループットとコンカレンシーを分析する。
PTFL-4.2.6 (K2) 性能テストスクリプトの基本構造を理解する。
PTFL-4.2.7 (K3) 計画書とロードプロファイルと整合性がとれた性能テストスクリプトを実装する。
PTFL-4.2.8 (K2) 性能テスト実行の準備に関わる活動を理解する。

4.3 実行

PTFL-4.3.1 (K2) 性能テストスクリプトを実行する際の主要な活動を理解する。

4.4 結果の分析とレポート

PTFL-4.4.1 (K4) 性能テストの結果とその意味するところを分析し、レポートする。

4.1 計画

4.1.1 性能テストの目的の導出

ステークホルダーには、ユーザー、およびビジネスまたは技術的背景を持つ人々が含まれる。ステークホルダーによっては、性能テストに関する目的が異なる場合がある。ステークホルダーは、目的、使用する用語、目的が達成されたかどうかを判断する基準を設定する。

性能テストの目的は、さまざまなタイプのステークホルダーに関連している。ユーザーに基づいた目的と技術的な目的を区別するのはよい方法である。ユーザーに基づいた目的は、主にエンドユーザーの満足度とビジネスゴールに焦点を当てる。一般的に、ユーザーは機能の種類や製品の提供方法にはあまり関心がない。ユーザーは、自分が必要とすることができることを望んでいる。

一方、技術的な目的は、運用面に焦点を当てており、システムの拡張性や、どのような条件で性能の低下が明らかになるかといった疑問に対する答えを提供する。

性能テストの主要な目的は、潜在的なリスクを識別し、改善の機会を見つけ、必要な変更を識別することである。

さまざまなステークホルダーから情報を集める際には、以下のような質問に答えられるべきである。

- 性能テストではどのようなトランザクションが実行され、どのくらいの平均応答時間が期待されるか？
- どのようなシステムメトリクス（メモリー使用量、ネットワークスループットなど）をキャプチャーし、どのような値を期待するか？
- 前回のテストサイクルと比較して、今回のテストではどのような性能向上が期待できるか？

4.1.2 性能テスト計画書

性能テスト計画書（PTP）は、性能テストを実施する前に作成される文書である。PTP は、関連するスケジュール情報を含むテスト計画書（[ISTQB_FL_SYL]参照）によって参照されるべきである。PTP は、性能テストが開始された後も更新され続ける。
PTP では、以下の情報を提供する必要がある。

目的
PTP の目的は、性能テストのゴール、戦略、手法を記述したものである。これにより、システムに負荷がかかったときの適切さと、リリース可能な状態になっているかという根幹に関わる問いに対する定量的な回答が可能になる。

テストの目的
テスト対象システム（SUT）が達成する性能効率性に関するテスト目的の全体は、ステークホルダーのタイプごとに列挙する（4.1.1 項参照）。

システム概要
SUT の概略は、性能テストパラメーターを測定するための背景を提供する。概要には、負荷がかかったときのテスト対象機能のハイレベルな説明を含めるべきである。

実施する性能テストのタイプ
実施する性能テストのタイプは、各タイプの目的の説明とともに列挙する（1.2 節参照）。

受け入れ基準
性能テストは、与えられたワークロードの下で、システムの応答性、スループット、信頼性、および／または拡張性を判断することを目的としている。一般的に、応答時間はユーザーの関心事であり、スループットはビジネスの関心事であり、リソースの使用率はシステムの関心事である。受け入れ基準は、関連するすべての測定値に対して設定され、必要に応じて以下の項目に関連する。

- 全体的な性能テストの目的
- サービスレベルアグリーメント（SLA）
- ベースライン値- ベースラインとは、現在の性能測定値と以前に達成された性能測定値を比較するために使用する一連のメトリクスのことである。これにより、特定の性能向上を実証、そして／またはテストの受け入れ基準の達成確認が可能になる。最初にベースラインを作成するためには、可能であれば、データベースのサニタイズされたデータを使うことが必要となることがある。

テストデータ
テストデータには、性能テストのために指定する必要のある広範なデータが含まれる。このデータには次のようなものがある。

- ユーザーアカウントデータ（例：同時ログイン可能なユーザーアカウント）

- ユーザー入力データ（例：ビジネスプロセスを実行するために、ユーザーがアプリケーションに入力するデータ）
- データベース（例：テストに使用するデータがあらかじめ入力されているデータベース）

テストデータの作成プロセスでは、以下の点に留意する必要がある。

- 本番データからのデータ抽出
- SUT に対するデータのインポート
- 新しいデータの作成
- 新しいテストサイクルの実行時にデータを復元するために使用できるバックアップの作成
- データのマスキングまたは匿名化。この方法は、個人を特定できる情報を含む本番データに使用され、一般データ保護規則（GDPR）では必須となっている。しかし、性能テストにおいては、データマスキングを行うと、実際の使用時に見られるようなデータ特性を持たなくなる可能性があるため、性能テストにとってのリスクが加わる。

システム構成

PTP のシステム構成の節は、以下の技術情報を含む。

- サーバー（Web、データベース、ロードバランサーなど）を含む、具体的なシステムアーキテクチャーの説明
- 複数の階層の定義
- コンピューティングハードウェア（例：CPU コア、RAM、SSD（ソリッドステートドライブ）、HDD（ハードディスクドライブ）のバージョンを含む具体的な詳細）
- ソフトウェア（アプリケーション、オペレーティングシステム、データベース、企業をサポートするために使用されるサービスなど）のバージョンを含む具体的な詳細
- SUT と一緒にに動作する外部システムおよびその構成とバージョン（例：NetSuite と統合した E コマースシステム）
- SUT のビルド／バージョン識別子

テスト環境

テスト環境は、多くの場合、本番環境を模倣した別の環境であるが、規模は小さくなる。PTP のこの節では、性能テストの結果を、より大規模な本番環境に適用する

ために、どのようにテスト環境のデータから推定するかを記載すべきである。システムによっては、本番環境でのテストが唯一の選択肢となる場合もあるが、その場合には、この種のテストの具体的なリスクについて検討しなければならない。

テストツールは、時にテスト環境そのものの外に存在し、システムコンポーネントとやりとりするために特別なアクセス権を必要とする場合がある。この場合、テスト環境と構成が考慮事項となる。

性能テストは、他のコンポーネントがなくても動作可能なシステムのコンポーネントの一部分を使って実施することもできる。この方法は、システム全体を使ってテストするよりもコストが安いことが多く、コンポーネントが開発されるとすぐに実施することができる。

テストツール
この節では、性能テストケースのスクリプト作成、実行、およびモニタリングに使用するテストツール（およびそのバージョン）について説明する（第 5 章参照）。このリストには通常、次のものが含まれる。

- ユーザートランザクションをシミュレーションするツール
- システムアーキテクチャー内の複数のポイント（ポイントオブプレゼンス）から負荷を発生させるツール
- システム性能をモニタリングするツール。 対象は前述の「システム構成」で説明したものを含む。

プロファイル
運用プロファイルは、システムの特定の使用方法について、アプリケーションを通じた再現可能なステップバイステップのフローを提供する。これらの運用プロファイルを集約すると、ロードプロファイル（一般的にはシナリオと呼ばれる）となる。プロファイルの詳細については、4.2.3 項を参照のこと。

関連するメトリクス
性能テストの実行中には、数多くの測定値やメトリクスを収集することができる（第 2 章参照）。しかし、あまりにも多くの測定値を取得すると、分析が困難になるだけでなく、アプリケーションの実際の性能に悪影響を及ぼす可能性がある。このような理由から、性能テストの目的を達成するために最も関連性の高い測定値やメトリクスを特定することが重要になる。

以下の表は、4.4 節で詳しく説明しているが、性能のテストとモニタリングのための典型的なメトリクスのセットを示している。性能に関するテスト目的は、必要に応じて、プロジェクトのこれらのメトリクスに対して定義されるべきである。

性能メトリクス	
タイプ	**メトリクス**
仮想ユーザーのステータス	# 成功 # 失敗
トランザクションの応答時間	最小 最大 平均 90%パーセンタイル
秒単位のトランザクション	# 成功/秒 # 失敗/秒 # 合計/秒
ヒット数（例：データベースや Web サーバー）	ヒット数/秒 ▪ 最小 ▪ 最大 ▪ 平均 ▪ 合計
スループット	ビット/秒 ▪ 最小 ▪ 最大 ▪ 平均 ▪ 合計
秒単位の HTTP レスポンス	レスポンス/秒 ▪ 最小 ▪ 最大 ▪ 平均 ▪ 合計 HTTP レスポンスコードごとのレスポンス

性能モニタリング	
タイプ	メトリクス
CPU 使用率	使用可能な CPU の割合
メモリー使用量	使用可能なメモリーの割合

リスク
リスクには、性能テストの一部として測定されていない領域や、性能テストの制限（例：シミュレーションできない外部インターフェース、不十分な負荷、サーバーをモニタリングできないことなど）が含まれる。また、テスト環境の制限によってもリスクが生じる可能性がある（例：不十分なデータ、スケールダウンされた環境）。より多くのリスクタイプについては、3.2 節および 3.3 節を参照のこと。

4.1.3 性能テストについてのコミュニケーション

テスト担当者は、すべてのステークホルダーに対して、性能テストのアプローチの背後にある理論的根拠と、（性能テスト計画書に詳述されているような）実施する活動を伝えられなければならない。このコミュニケーションで扱うテーマは、ステークホルダーが「ビジネス／ユーザー面」に関心を持っているか、より「技術／運用面」に関心を持っているかによって大きく異なる可能性がある。

ビジネスに重心を置いたステークホルダー
ビジネスに重心を置いたステークホルダーとのコミュニケーションでは、以下の要素を考慮する必要がある。

- ビジネスに重心を置いたステークホルダーは、機能的品質特性と非機能的品質特性の区別にあまり関心がない。
- ツール、スクリプト作成、負荷生成などの技術的な問題は、一般的には二の次になる。
- プロダクトリスクと性能テストの目的の関連性が明確に示されていなければならない。
- ステークホルダーは、計画した性能テストのコストと、本番環境の条件と比較して、性能テストの結果が本番を再現できている度合いのバランスを認識しなければならない。
- そのテストケースは繰り返すのが難しいのか、それとも最小限の労力で繰り返すことができるのか？といった、計画した性能テストの再現性を伝えなければならない。

- プロジェクトリスクを伝えなければならない。これには、テストケースのセットアップに関する制約や依存関係、インフラ要件（ハードウェア、ツール、データ、帯域幅、テスト環境、リソースなど）や主要スタッフへの依存関係などが含まれる。
- コスト、タイムスケジュール、マイルストーンを含む大まかな計画書とハイレベルな活動を伝えなければならない（4.2 および 4.3 節参照）。

技術に重心を置いたステークホルダー

技術に重心を置いたステークホルダーとのコミュニケーションでは、以下の要素を考慮する必要がある。

- 計画した必要なロードプロファイルを生成するためのアプローチを説明し、技術的なステークホルダーの期待される関与を明確にしなければならない。
- 性能テストケースのセットアップと実行の詳細なステップを説明し、テストとアーキテクチャーに関わるリスクとの関係を示さなければならない。
- 繰り返し可能な性能テストケースとするために必要なステップを伝えなければならない。これには、技術的な懸念事項だけでなく、組織的な側面（主要スタッフの参加など）を含むことがある。
- テスト環境を共有する場合、テスト結果に悪影響を及ぼさないように、性能テストケースを実行するスケジュールを伝えなければならない。
- 性能テストを本番環境にて実行する必要がある場合、実ユーザーに影響を与える可能性があるため、その軽減策を伝え、受け入れてもらわなければならない。
- 技術的なステークホルダーは、自分のタスクとそのスケジュールを明確にしなければならない。

4.2 分析、設計、実装

4.2.1 代表的な通信プロトコル

通信プロトコルは、コンピューターやシステム間の通信ルールのセットを定義するものである。システムの特定の部分を対象としたテストケースを適切に設計するには、プロトコルを理解する必要がある。

通信プロトコルは、OSI（Open Systems Interconnection）モデルの階層で記述されることが多いが（ISO/IEC 7498-1 参照）、プロトコルによってはこのモデルから外れるものもある。性能テストでは、第 5 層（セッション層）から第 7 層（アプリケ

ーション層）までのプロトコルが最も一般的に使用される。一般的なプロトコルには次のようなものがある。

- データベース - ODBC、JDBC、その他ベンダー固有のプロトコル
- Web - HTTP、HTTPS、HTML
- Web サービス- SOAP, REST

一般的に、性能テストで最も重視される OSI 層のレベルは、テスト対象のアーキテクチャーのレベルに関連している。例えば、低レベルの組込みアーキテクチャーをテストする場合、OSI モデルの下位番号の層が主に重視される。

性能テストで使用されるその他のプロトコルは以下の通り。

- ネットワーク - DNS、FTP、IMAP、LDAP、POP3、SMTP、Windows ソケット、CORBA
- モバイル- TruClient、SMP、MMS
- リモートアクセス- Citrix ICA, RTE
- SOA- MQSeries、JSON、WSCL

性能テストケースは、個々のシステムコンポーネント（Web サーバー、データベースサーバーなど）に対して実行することも、エンドツーエンドテストによってシステム全体に対して実行することもできるため、システム全体のアーキテクチャーを理解することが重要である。クライアントサーバーモデルで構築された伝統的な 2 層アプリケーションでは、「クライアント」が GUI や主要なユーザーインターフェースとして、「サーバー」がバックエンドのデータベースとして仕様化されている。このようなアプリケーションでは、データベースにアクセスするために ODBC などのプロトコルを使用する必要がある。Web ベースのアプリケーションや多層アーキテクチャーの進化に伴い、多くのサーバーが、最終的にユーザーのブラウザーに表示される情報の処理に関与している。

テストの対象となるシステムの部分に応じて、使用する適切なプロトコルを理解する必要がある。例えば、ブラウザーからのユーザーのアクティビティを模倣するエンドツーエンドテストを実施する必要がある場合は、HTTP/HTTPS のような Web プロトコルを使用する。こうすることで、GUI とのやりとりを行わず、テストケースではバックエンドサーバーの通信とアクティビティに注力することが可能になる。

4.2.2 トランザクション

トランザクションとは、開始時点から 1 つ以上のプロセス（リクエスト、操作、操作プロセス）が完了するまでにシステムが実行する一連のアクティビティのことである。トランザクションの応答時間は、システムの性能を評価する目的で測定する

ことができる。性能テストでは、この測定値を修正や最適化が必要なコンポーネントを特定するために使う。

シミュレーションされたトランザクションには、実際のユーザーが行動を起こす（例えば、「SEND」ボタンを押す）タイミングをよりよく反映させるためのシンクタイムを含めることができる。トランザクションの応答時間にシンクタイムを加えたものが、そのトランザクションの経過時間となる。

性能テスト中に収集されたトランザクションの応答時間は、システムにかけた異なる負荷の下でこの測定値がどのように変化するかを示している。分析結果では負荷による劣化が見られない場合もあれば、他の測定結果では深刻な劣化が見られる場合もある。負荷をランプアップし、基礎となるトランザクション時間を測定することで、劣化の原因を 1 つまたは複数のトランザクションの応答時間と関連付けることが可能になる。

また、トランザクションを入れ子にすることで、個々のアクティビティと集約されたアクティビティを測定することができる。これは、例えば、オンライン注文システムの性能効率性を理解する際に役立つ。テスト担当者は、注文プロセスにおける個別のステップ（例：商品の検索、商品のカートへの追加、商品の支払い、注文の確認）に加えて、注文プロセス全体も測定したいと考えるかもしれない。トランザクションを入れ子にすることで、1 回のテストで両方の情報を収集することができる。

4.2.3 運用プロファイルの特定

運用プロファイルは、ユーザーや他のシステムコンポーネントなどと、アプリケーションとのやりとりの明確なパターンを仕様化する。1 つのアプリケーションに対して複数の運用プロファイルを指定することができる。これらを組み合わせて、特定の性能テスト目的を達成するための望ましいロードプロファイルを作成することができる（4.2.4 項参照）。

この項では、運用プロファイルを特定するための以下の主要なステップについて説明する。

1. 収集すべきデータの特定
2. 1 つまたは複数のソースを使ってデータを収集する
3. データを評価して運用プロファイルを構築する

データの特定

ユーザーがテスト対象のシステムとやりとりをする場合、ユーザーの運用プロファイル（つまり、どうシステムとやりとりをするか）をモデル化するために、以下のデータを収集または見積もりする。

- さまざまなタイプのユーザーペルソナとその役割（例：標準ユーザー、登録メンバー、管理者、特定の権限を持つユーザーグループ）。
- これらのユーザー/役割によって実行されるさまざまな一般的なタスク（例：情報を得るために Web サイトを閲覧する、特定の製品を探すために Web サイトを検索する、該当の役割に特化した活動を行う）。これらのタスクは、一般的に高い抽象度でモデル化するのが最適であることに注意する（例：ビジネスプロセスや主要なユーザーストーリーのレベル）。
- 一定の期間における単位時間あたりで見積もられた各役割／タスクのユーザー数。この情報は、その後のロードプロファイルの作成にも役立つ（4.2.4 項参照）。

データの収集

上記のようなデータは、さまざまな情報源から収集することができる。

- プロダクトオーナー、セールスマネージャー、（潜在的な）エンドユーザーなどのステークホルダーとのインタビューやワークショップの実施。このような話し合いにより、ユーザーの主な運用プロファイルが明らかになり、「このアプリケーションは誰のためのものか」という基本的な質問に対する答えが得られることがよくある。
- 機能仕様と要件（入手可能な場合）は、意図された使用パターンに関する貴重な情報源であり、ユーザーのタイプとその運用プロファイルの特定にも役立つ。機能仕様がユーザーストーリーとして表現されている場合、標準的なフォーマットにより、ユーザーのタイプを直接特定することができる（例：<ユーザーのタイプ>として, 私は<特徴や性能>をしたい , なぜなら < 得られるメリット>だからだ）。同様に、UML のユースケース図や説明文では、ユースケースの「アクター」を特定する。
- 類似のアプリケーションから得られた使用データやメトリクスを評価することで、ユーザータイプの特定を支援し、予想ユーザー数の初期値を得ることができる。自動的にモニターされるデータ（例：Web マスターの管理ツール）にアクセスすることを推奨する。これには、現在の運用システムの更新が予定されている場合に、その使用状況から得られるモニタリングのログやデータが含まれる。
- アプリケーションであらかじめ定義されたタスクを実行する際のユーザーの振る舞いをモニタリングすることで、性能テストでモデル化すべき運用プロファイルの種類についての洞察を得ることができる。このタスクは、（ユーザビリティラボが利

用可能な場合は特に）計画されている使用性テストの実施と調整することをお勧めする。

運用プロファイルの構築

ユーザーの運用プロファイルを特定して構築するためには、以下のステップに従う。

- トップダウン方式を採用する。比較的単純で大まかな運用プロファイルを最初に作成し、性能テストの目的を達成するために必要な場合のみ、さらに詳細化する。（4.1.1 項参照）。
- 特定のユーザープロファイルは、頻繁に実行されるタスクを含む場合、異なるシステムコンポーネント間で重要（高リスク）または頻繁なトランザクションを必要とする場合、または大量のデータの転送を必要とする可能性がある場合に、性能テストと関連する特定のユーザープロファイルとして選択することがある。
- 運用プロファイルは、ロードプロファイルの作成に使う前に、主要なステークホルダーによるレビューを行い、洗練する（4.2.4 項参照）。

テスト対象システムは、必ずしもユーザーからの負荷を受けるわけではない。運用プロファイルは、以下のタイプのシステムの性能テストにも必要となる場合がある（このリストはすべてを網羅しているわけではないことに注意が必要である）。

オフラインのバッチ処理システム

ここでは主に、バッチ処理システムのスループット（4.2.5 項参照）と、所定の期間内に完了する能力に焦点を当てている。運用プロファイルは、バッチ処理に要求される処理の種類に焦点を当てている。例えば、株式取引システム（通常、オンラインおよびバッチベースの取引処理を含む）の運用プロファイルには、支払い取引、クレデンシャル情報の確認、および特定の種類の株式取引に関する法的条件の遵守の確認に関するものが含まれる。これらの運用プロファイルはそれぞれ、株式のバッチ処理全体の中で異なるパスを通ることになる。上記で説明したオンラインのユーザーを前提にしたシステムでの運用プロファイルを特定するステップはバッチ処理の文脈にも適用できる。

システムオブシステムズ

複合システム環境内のコンポーネント（組込みの場合もある）は、他のシステムやコンポーネントからのさまざまな種類の入力に応答する。テスト対象システムの性質にもよるが、サプライヤーのシステムから提供される入力の種類を効果的に表現するために、複数の異なる運用プロファイルのモデリングが必要になる場合がある。

これには、システムアーキテクトと共同による、システムやインターフェースの仕様に基づいた詳細な分析（バッファやキューなど）を含むことがある。

4.2.4 ロードプロファイルの作成

ロードプロファイルは、本番でテスト対象のコンポーネントやシステムにて行うと思われる動作を仕様化するものである。ロードプロファイルは、指定した数のインスタンスで構成される。インスタンスは、特定の期間にわたり、定義済みの運用プロファイルのアクションを行う。インスタンスがユーザーである場合、「仮想ユーザー」という用語を一般的に適用する。

現実に即していて繰り返し可能なロードプロファイルを作成するために必要な主な情報は以下の通り。

- 性能テストの目的（例：ストレスがかかる負荷の下でのシステム挙動の評価）
- 個々の利用パターンを正確に表す運用プロファイル（4.2.3 項参照）
- スループットやコンカレンシーに関する既知の問題（4.2.5 項参照）
- 運用プロファイルを実行する量および時間分布。 SUT にて所望の負荷がかかるようにする。典型的な例は以下の通り。
    - ランプアップ：段階的に負荷を高めていくこと（例：1 分間に 1 人の仮想ユーザーを追加するなど
    - ランプダウン：段階的に負荷を下げること
    - ステップ：瞬間的な負荷の変化のこと（例：5 分ごとに 100 人の仮想ユーザーを追加するなど）
    - 事前定義分布：（例：日次および季節的なビジネスサイクルを模したボリュームなど）

以下の例は、テスト対象システムにストレス条件（システムが処理できる想定最大値、あるいはそれ以上）を発生させることを目的としたロードプロファイルの構築を示している。

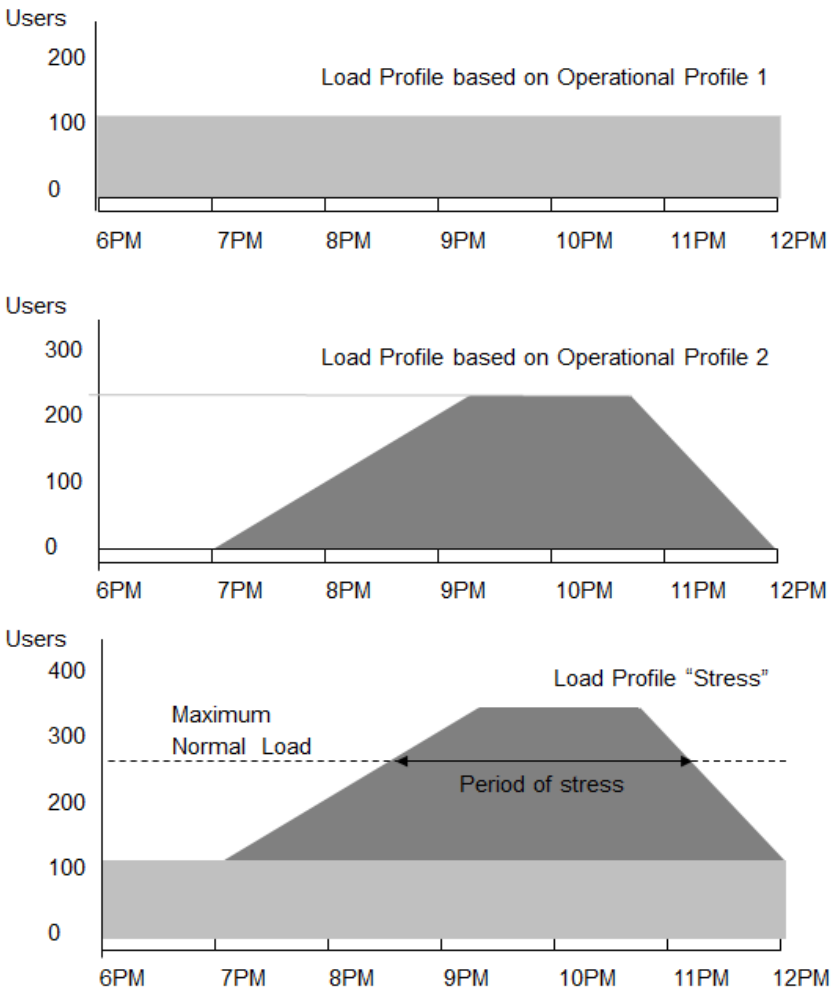

図 1：「ストレス」ロードプロファイルの構築例

最初の図は、100 人の仮想ユーザーをステップしたロードプロファイルを示している。これらのユーザーは、テストの全期間において、Operational Profile 1 で定義されたアクティビティを実行する。これは、バックグラウンドの負荷を表す多くの性能ロードプロファイルの典型である。

2 番目の図は、220 人まで仮想ユーザーをランプアップし、2 時間維持した後にランプダウンするロードプロファイルを示している。各仮想ユーザーは、Operational Profile 2 で定義されたアクティビティを実行する。

最後の図は、上述の 2 つを組み合わせた結果としてのロードプロファイルを示している。テスト対象のシステムには3時間のストレスがかかる。更なる例については、[Bath14]を参照。

4.2.5 スループットとコンカレンシーの分析

ワークロードの異なる側面であるスループットとコンカレンシーを理解することは重要である。運用プロファイルとロードプロファイルを適切にモデル化するには、両方の側面を考慮すべきである。

システムスループット
システムスループットとは、単位時間内にシステムが処理する特定のタイプのトランザクション数を示す尺度である。例えば、1 時間あたりの注文数や、1 秒あたりの HTTP リクエスト数などである。システムスループットは、ネットワーク上を移動するデータ量であるネットワークスループットとは区別する必要がある（2.1 節参照）。

システムスループットは、システムにかかる負荷の定義となる。残念ながら、システムとのやりとりの負荷を定義するのに、スループットではなく、同時ユーザー数を使うことがよくある。これは部分的には正しいと言える。その数を見つけるのが容易な場合が多いということが理由であるが、負荷テストツールが同様の方法で負荷を定義することも理由の 1 つである。運用プロファイル（各ユーザーが行うこととその頻度、つまり1ユーザーあたりのスループットでもある）を定義しなければ、ユーザー数は負荷の尺度としてよいものとはならない。例えば、500 人のユーザーが短いクエリーを毎分実行する場合、1 時間あたり 30,000 クエリーのスループットとなる。同じ 500 人のユーザーが同じクエリーを実行していても、1 時間に 1 回であれば、スループットは 1 時間あたり 500 クエリーとなる。つまり、同じ 500 人のユーザーがいても、負荷に 60 倍の差があり、システムに必要なハードウェアにも少なくとも 60 倍の差があるということである。

ワークロードのモデル化は、通常、仮想ユーザー（実行スレッド）の数とシンクタイム（ユーザーアクション間の遅延）を考慮して行われる。しかし、システムスループットは処理時間によっても定義され、負荷の増加に伴って処理時間が長くなる可能性がある。

システムスループット＝[仮想ユーザー数] ÷（[処理時間]＋[シンクタイム]）

そのため、処理時間が長くなると、他の条件が同じでもスループットが大幅に低下することがある。

バッチ処理システムをテストする際には、システムスループットが重要なポイントとなる。この場合、スループットは、通常、所定の時間枠（例えば、夜間のバッチ処理ウィンドウ訳注）内で達成できるトランザクション数に従って測定される。
JSTQB 訳注-ウィンドウとはバッチ処理実行に許された時間枠のこと

コンカレンシー
コンカレンシーとは、同時/並列に実行されるスレッドの数を示す尺度である。システムとのやりとりでは、同時/並列ユーザーの数を表すこともある。コンカレンシーは通常、負荷テストツールで仮想ユーザーの数を設定することでモデル化する。
コンカレンシーは重要な尺度の 1 つである。コンカレンシーは並行して実行されるセッションの数を表しており、それぞれのセッションが個別のリソースを使用する可能性がある。スループットが同じであっても、使用されるリソースの総量はコンカレンシーに応じて異なる可能性がある。典型的なテスト用にセットアップしたシステムは、（待ち行列理論の観点では）閉じたシステムであり、システム内のユーザー数は設定されている（固定されたユーザー数）。もし、閉じたシステムですべてのユーザーがシステムからのレスポンスを待つのであれば、新しいユーザーはその間システムに入ることはできない。多くの公開されているシステムは開かれたシステムであり、システムにいるユーザー全員がシステムのレスポンスを待っている間だとしても、新しいユーザーが常にシステムに入ってくる。

4.2.6 性能テストスクリプトの基本構造

性能テストスクリプトは、テスト対象のシステム（システム全体またはそのコンポーネントの 1 つ）の負荷に影響するユーザーまたはコンポーネントのアクティビティをシミュレーションすべきである。このスクリプトは、適切な順序で、所定のペースでサーバーへのリクエストを送り始める。

性能テストスクリプトを作成する最適な方法は、負荷生成のアプローチによって異なる（4.1 節）。

- 従来からの方法は、クライアントとシステムやコンポーネント間の通信をプロトコルレベルで記録し、スクリプトのパラメーター化とドキュメント化を行った後に再生する方法である。パラメーター化の結果、拡張性と保守性に優れたスクリプトが完成するが、パラメーター化の作業には時間がかかる場合がある。
- GUI レベルでの記録では、テスト実行ツールで 1 つのクライアントの GUI アクションをキャプチャーし、そのスクリプトを負荷生成ツールで実行して複数のクライアントを表現するのが一般的である。
- プログラミングは、プロトコルリクエスト（HTTP リクエストなど）、GUI アクション、または API コールを使用して行われる。プログラミングスクリプトの場合、実際

のシステムに送信されるリクエストと受信されるリクエストの正確な順序を決定する必要があるが、これは簡単なことではない。

通常、スクリプトは、（拡張機能を備えた一般的なプログラミング言語または特殊な言語で記述された）コードの 1 つまたは複数のセクション、またはオブジェクトであり、ツールによって GUI でユーザーに表示されることがある。いずれの場合も、スクリプトには、負荷を作成するサーバーリクエスト（HTTP リクエストなど）と、これらのリクエストをどのように実行するか（どのような順番で、どのようなタイミングで、どのようなパラメーターで、何をチェックすべきかなど）を指定するプログラミングロジックが含まれる。ロジックが高度になればなるほど、強力なプログラミング言語が必要になる。

全体の構成
多くの場合、スクリプトには、初期化セクション（メインパートのためにすべての準備をする）、複数回実行される場合があるメインセクション、クリーンアップセクション（テストを適切に終了するために必要なステップを実行する）がある。

データ収集
応答時間を収集するためには、スクリプトにタイマーを追加して、リクエストまたはリクエストの組み合わせにかかる時間を測定すべきである。時間計測の対象になるリクエストは、論理的な作業の意味のある単位と一致すべきである。例えば、注文にアイテムを追加したり、注文を送信したりするビジネストランザクションなどの単位である。

具体的に何を計測しているかを知ることは重要である。プロトコルレベルのスクリプトの場合はサーバーとネットワークの応答時間のみであるが、GUI スクリプトの場合はエンドツーエンドの時間を測定している（ただし、使用する技術によって具体的に何を測定するかは異なる）。

結果検証とエラー処理
スクリプトの重要な部分の 1 つとして、結果の検証とエラー処理がある。どんなに優れた負荷テストツールでも、デフォルトのエラー処理（HTTP リクエストのリターンコードをチェックするなど）は最小限に限られる傾向があるので、リクエストが実際に何を返すのかを検証するためのチェックを追加することをお勧めする。また、エラーが発生した場合に何らかのクリーンアップが必要な場合も、手動で実装する必要があるかもしれない。スクリプトが想定通りの動作をしているかどうかを間接

的な方法で確認するのがよい方法である。例えば、データベースをチェックして適切な情報が追加されているかどうかを確認するなどの方法である。

スクリプトには、サーバーへのリクエストがいつ、どのように行われるかというルールを指定する他のロジックが含まれることがある。例えば、同期ポイントを設定することで、スクリプトがそのポイントでイベントを待ってから処理を進めるように指定することができる。同期ポイントは、特定のアクションが同時に実行されるようにしたり、複数のスクリプト間で作業を調整したりするために使用できる。

性能テストスクリプトはソフトウェアであるため、性能テストスクリプトを作成することは、1 つのソフトウェア開発活動である。性能テストスクリプトの作成には、品質保証とテストを含めるべきである。テストでは、入力データの全範囲でスクリプトが期待通りに動作することを検証する。

4.2.7 性能テストスクリプトの実装

性能テストスクリプトは、PTP およびロードプロファイルに基づいて実装する。実装の技術的な詳細は、使用するアプローチやツールによって異なるが、全体的なプロセスは同じである。性能スクリプトは、統合開発環境（IDE）やスクリプトエディターを使用して、ユーザーやコンポーネントの動作をシミュレーションするために作成する。通常、スクリプトは 1 つの特定の運用プロファイルをシミュレーションするために作成する（ただし、条件文を用いて複数の運用プロファイルを 1 つのスクリプトにまとめることができる場合もある）。

リクエストの順序が決まると、アプローチに応じてスクリプトを記録したり、プログラミングしたりすることができる。通常、記録は実際のシステムを正確にシミュレーションすることを保証し、プログラミングは適切なリクエストの順序の知識に依存する。

プロトコルレベルでの記録を行う場合、記録後の重要なステップとして、処理内容を定義するために記録されたすべての内部識別子を置き換えることが多い。これらの識別子は、あるリクエストのレスポンスから抽出した適切な値で、実行ごとに変更可能な変数にしなければならない（例えば、ログイン時に取得され、その後のすべてのトランザクションで与えなければならないユーザー識別子など）。これはスクリプトのパラメーター化の一部であり、「相関(correlation)」と呼ばれることもある。この文脈では、相関という言葉は、統計学で使われる場合（2 つ以上の物事の関係を意味する）とは異なる意味を持っている。機能が豊富な負荷テストツールでは、相関を自動で行うことがある。しかし、より複雑なケースでは、手動で相関を行っ

たり、新しい相関ルールを追加したりする必要があるかもしれない。記録されたスクリプトが再生できない主な理由は、相関が正しくないか、相関がないことである。

同じユーザー名で複数の仮想ユーザーを実行し、同じデータセットにアクセスすると、誤解を招くような結果を容易に得られる（通常、記録したスクリプトを必要最低限の相関以外の変更をせずに再生するときに起こる）。データが完全にキャッシュ（高速アクセスのためにディスクからメモリーにコピー）されていれば、（そのようなデータがディスクから読み込まれる場合）、実稼働時よりもはるかによい結果が得られる。また、同じユーザーやデータを使用すると（例えば、ユーザーがデータを更新しているときにデータがロックされた場合）、コンカレンシーの問題も発生する。ソフトウェアは、次のユーザーが更新のためにデータをロックする前にロックが解除されるのを待つために、本番環境よりもはるかに悪い結果が得られる。

そのため、スクリプトやテストハーネスはパラメーター化し、各仮想ユーザーが適切なデータセットを使用するようにすべきである（つまり、固定または記録されたデータは、リストの選択可能な値に置き換えるべきである）。ここで言う「適切な」とは、キャッシングやコンカレンシーの問題を回避するのに十分な違いを意味しており、システム、データ、テスト要件に応じて異なる。この更なるパラメーター化は、システム内のデータと、システムがこのデータを扱う方法に依存するため、多くのツールがここをサポートしているものの、通常は手動で行う。

例えば、注文を作成する際に注文名が一意でなければならない場合など、テストが複数回動作するために一部のデータをパラメーター化する必要がある。注文名がパラメーター化されていなければ、既存の (記録された) 名前で注文を作成しようとした時点でテストは失敗する。

4.2.5 項で考察しているように、運用プロファイルと一致させるためには、適切なリクエスト数／スループットを生成するために、シンクタイムを挿入、そして／または（記録されている場合）調整すべきである。

別々の運用プロファイルのスクリプトが作成されると、それらはロードプロファイル全体を実装するシナリオにまとめられる。ロードプロファイルでは、各スクリプトを使用して、いつ、どのようなパラメーターで、何人の仮想ユーザーを開始するかをコントロールする。具体的な実装の詳細は、特定の負荷テストツールまたはテストハーネスに依存する。

4.2.8 性能テストケース実行のための準備

性能テストケースの実行準備のための主な活動は以下の通り。

- テスト対象システムのセットアップ
- 環境のデプロイ
- 負荷生成ツール、およびモニタリングツールのセットアップと、必要な情報が収集できることの確認

テスト環境が可能な限り本番環境に近いものであることが重要である。それが不可能な場合は、その違いと、どのようにテスト結果から本番環境を予測するのかを明確に理解しなければならない。理想的には、実際の本番環境とデータを使用するべきだが、スケールダウンした環境にて行うテストでも、多くの性能リスクを軽減する手助けとなる場合がある。

性能と環境の関係は非線形であることを覚えておくことが重要である。つまり、テスト環境が本番の標準的な環境から離れれば離れるほど、本番の性能を正確に予測することは難しくなる。テストシステムが本番と同等でなくなるほど、予測の信頼性が低くなり、リスクレベルが高まる。

テスト環境で最も重要な部分は、データ、ハードウェアとソフトウェアの構成、ネットワークの構成である。データのサイズと構造は、負荷テストの結果に大きく影響する。小さなデータのサンプルセットや、データの複雑さが異なるサンプルセットを性能テストケースに使用すると、誤解を招くような結果になる可能性がある。特に、本番システムで大きなデータセットを使用する場合は注意が必要である。実際にテストを行う前に、データサイズがどの程度性能に影響するかを予測することは困難である。テストデータのサイズや構造が本番データに近ければ近いほど、テスト結果の信頼性は高くなる。

テスト中にデータが生成または変更される場合、システムが適切な状態であることを確認するために、次のテストサイクルの前に元のデータを復元する必要があるかもしれない。

何らかの理由でシステムの一部やデータの一部が性能テストケースで利用できない場合は、回避策を講じるべきである。例えば、クレジットカードの処理を行うサードパーティのコンポーネントを代替し、模倣するためスタブを実装する場合がある。このようなプロセスは、しばしば「サービス仮想化」と呼ばれ、このプロセスを支援するための特別なツールがある。テスト対象システムを分離できるようにするために、このようなツールを使用することを強くお勧めする。

環境をデプロイするには、さまざまな方法がある。例えば、以下のような方法が考えられる。

- 従来からの内部（および外部）テストラボ
- IaaS （Infrastructure as a Service）を利用した環境としてのクラウドで、システムの一部または全部をクラウド上にデプロイしたもの
- SaaS（Software as a Service）を利用した環境としてのクラウドで、ベンダーが負荷テストサービスを提供する場合

具体的なゴールやテストするシステムに応じて、あるテスト環境が他の環境よりも望ましい場合がある。例えば

- 性能の改善（性能の最適化）の効果をテストするには、変更によって生じたわずかな変動を確認するために、隔離されたラボ環境を使用する方がよい場合がある。
- 本番環境全体をエンドツーエンドで負荷テストし、システムが大きな問題なく負荷を処理できることを確認するには、クラウドやサービスからのテストが適している場合がある（この方法は、クラウドからアクセスできる SUT に対してのみ有効であることに注意する必要がある）
- 性能テストの時間が限られている場合にコストを最小限に抑えるためには、クラウド上にテスト環境を構築することがより経済的な解決策となる場合がある。

どのようなデプロイ方法であっても、テストの目的や計画書に合わせてハードウェアとソフトウェアの両方を構成するべきである。環境が本番環境と一致する場合は、同じように構成するべきである。しかし、違いがある場合には、その違いに合わせて構成を調整する必要があるかもしれない。例えば、テストマシンの物理メモリーが本番マシンよりも少ない場合、メモリーのページングを避けるために、ソフトウェアのメモリーパラメーター（Java のヒープサイズなど）を調整する必要があるかもしれない。

グローバルシステムやモバイルシステムでは、ネットワークの適切な設定やエミュレーションが重要である。グローバルシステム（ユーザーや処理が世界中に分散しているシステム）では、ユーザーがいる場所にロードジェネレーターを配置することが 1 つのアプローチとなる。モバイルシステムでは、使用されるネットワークの種類の違いから、ネットワークエミュレーションが最も現実的な選択肢となる。負荷テストツールの中には、ネットワークエミュレーションツールが組み込まれているものもあれば、ネットワークエミュレーション用のスタンドアロンツールもある。

負荷生成ツールは、適切にデプロイすべきであり、モニタリングツールは、テストケースに必要なすべてのメトリクスを収集できるよう設定すべきである。メトリクスのリストはテストの目的によって異なるが、すべてのテストケースにおいて、少なくとも基本的なメトリクスは収集するのがよい（2.1.2 項参照）。

負荷、特定のツール／負荷生成アプローチ、およびマシン構成によっては、複数の負荷生成をするマシンが必要となる場合がある。テスト用のセットアップを検証するために、負荷生成に関わるマシンもモニタリングすべきである。これは、ロードジェネレーターのどれかの遅延で負荷が適切に維持されない状況の回避に役立つ。

テスト用のセットアップや使用するツールに応じて、適切な負荷を作り出すように負荷テストツールを設定する必要がある。例えば、特定のブラウザーエミュレーションのパラメーターを設定したり、IP スプーフィング（仮想ユーザーのそれぞれが異なる IP アドレスを持つようシミュレーションすること）を使用したりすることができる。

テストケースを実行する前に、環境とテスト用のセットアップを検証しなければならない。これは通常、コントロールされたテストケースのセットを実施し、テストケースの出力結果を検証するとともに、モニタリングツールが重要な情報を追跡していることを確認することで行われる。

テストが設計通りに動作しているかどうかを確認するためには、ログ分析やデータベースの内容の確認など、さまざまな技法が用いられる。テストの準備には、必要な情報をログに記録できるか、システムが適切な状態にあるかなどを確認することが含まれる。例えば、テストによってシステムの状態が大きく変化する場合（データベースの情報を追加・変更する場合）、テストを繰り返す前にシステムを元の状態に戻すことが必要になる場合がある。

4.3 実行

性能テストの実行には、（通常、与えられたシナリオに従って起動する性能テストスクリプトとして実装する）ロードプロファイルに従って SUT に対して負荷を生成すること、環境のすべての部分をモニタリングすること、およびテストに関連するすべての結果と情報を収集して保持することが含まれる。通常、高度な負荷テストツール／ハーネスは、（もちろん、適切な設定を行った後に）これらのタスクを自動的に実行する。これらのツールは一般的に、テスト中に性能データをモニタリングし、必要な調整を可能にするコンソールを提供する（5.1 節参照）。しかしながら、

使用するツール、SUT、および実行する特定のテストケースによっては、いくつかの手動ステップが必要となる場合がある。

性能テストケースは通常、システムの定常状態、つまりシステムの動作が安定しているときに焦点を当てる。例えば、すべてのシミュレーションされたユーザー／スレッドが起動し、設計通りに動作しているときである。負荷が変化しているとき（例えば、新しいユーザーが追加されたとき）は、システムの動作が変化しているので、テスト結果をモニタリング・分析することが難しくなる。定常状態になるまでの段階を「ランプアップ」、テストが終了するまでの段階を「ランプダウン」と呼ぶことがある。

システムの動作が変化するような過渡的な状態をテストすることが重要な場合がある。これは、例えば、多数のユーザーの同時書き込みやスパイクテストなどが該当する。過渡的な状態をテストする際には、慎重なモニタリングと慎重な結果分析の必要性を理解することが重要である。なぜなら、平均値をモニタリングするような標準的なアプローチでは、非常に誤解を招く結果となる可能性があるからである。

ランプアップにおいては、負荷状態を段階的に変化させて、着実に増加する負荷がシステムの応答に与える影響をモニタリングすることが望ましい。これにより、ランプアップに十分な時間が割り当てられていること、システムが負荷を処理できることを確認できる。定常状態に到達した後は、負荷とシステムの応答の両方が安定しており、（常に存在する）不規則な変動が大きくないことをモニタリングするのがよい方法である。

（負荷をかけているときに故障が発生しても）システムに懸念事項がないことを明らかにするには、故障をどのように処理するかを明確にしておくことが重要となる。例えば、故障が発生したときには、ユーザーはログアウトして、そのユーザーに関連するすべてのリソースは解放することが重要かもしれない。

モニタリングが負荷テストツールに組み込まれており、適切に設定されている場合には、通常、モニタリングはテスト実行と同時に開始する。しかし、スタンドアロンのモニタリングツールを使用する場合は、モニタリングを個別に開始して必要な情報を収集し、テスト結果の分析と合わせてその後の分析を実施できるようにすべきである。これはログ分析についても同様である。特定のテスト実行サイクルに関連するすべての情報を見つけることができるように、使用するすべてのツールを時間的に同期させることが重要である。

テスト実行は、性能テストツールのコンソールやリアルタイムのログ分析を使ってモニタリングし、テストケースと SUT の両方に懸念事項やエラーがないかをチェックすることが多い。これは、大規模なテストケースを無駄に動かし続けることを避けることに役立つ。大規模なテストケースは、うまくいかない場合（例えば、故障が発生したり、コンポーネントが失敗したり、または生成された負荷が低すぎたり高すぎたりした場合）、他のシステムにまで影響を与える可能性がある。このようなテストケースを動かすには大きなコストがかかり、テストケースが期待される動作から逸脱した場合には、テストケースを止めたり、または性能テストケースやシステム構成をその場で調整したりする必要があるかもしれない。

プロトコルレベルで直接通信をする負荷テストケースを検証する技法の1つとして、負荷テストケースを動かしているのと並行して、複数の GUI レベルの（機能的な）スクリプトを動かしたり、同様の運用プロファイルを手動で動かしたりする方法がある。この方法により、テスト中にレポートされた応答時間が GUI レベルで手動計測した応答時間と、クライアント側で費やされる時間分だけ異なることがチェックできる。

（例えば、3.4 節 で述べたように、 継続的インテグレーションの一部として）自動で性能テストを動かす場合、手動でのモニタリングや介入ができないため、チェックを自動的に行わなければならない。この場合、テストのセットアップは、逸脱や問題を認識して、（通常は、テストケースを適切に完了しながら）警告を発するべきである。このアプローチは、システムの動作がよく理解されているリグレッションテストにおける性能テストでは比較的容易に実装できるが、テスト中に動的な調整が必要となるような探索的な性能テストや大規模で高価な性能テストではより困難な場合がある。

4.4 結果の分析とレポート

4.1.2 項 では、性能テスト計画書におけるさまざまなメトリクスについて説明した。これらを前もって定義することで、各テスト実行で何を測定しなければならないかが決まる。テストサイクルの完了後、定義されたメトリクスに基づいてデータを収集する必要がある。

データを分析する際には、まず性能テストの目的と比較する。挙動が理解できれば、推奨されるアクションを含む有意義なサマリーレポートを提供する結論を導き出すことができる。これらのアクションには、物理的なコンポーネント（ハードウェア、ルーターなど）の変更、ソフトウェアの変更（アプリケーションやデータベースコ

ールの最適化など）、ネットワークの変更（ロードバランシング、ルーティングなど）が含まれる。

典型的には以下のデータを分析する。

- **シミュレーションされた（例：仮想）ユーザーの状況。** これは最初に調査する必要がある。通常、すべてのシミュレーションされたユーザーは、運用プロファイルで指定されたタスクを達成することができると期待されている。このシミュレーションされたユーザーのアクティビティの何らかの中断は、実際のユーザーが経験することと同じことになる。何らかのエラーの発生は、他の性能データに影響を与える可能性があるため、最初にすべてのユーザーのアクティビティが完了したことを確認することが非常に重要である。

- **トランザクションの応答時間。** これは、最小値、最大値、平均値、パーセンタイル（例：90 パーセンタイル）など、複数の方法で測定できる。最小値と最大値は、システム性能の極端な値を示している。平均値は、必ずしも数学的な平均値以外の意味を示すものではなく、外れ値によって歪んでしまうこともある。90 パーセンタイルは、大多数のユーザーが特定の性能しきい値を達成していることを示すため、性能ゴールとしてよく使用される。性能の目的を 100%遵守することを求めるのは、必要とされるリソースが大きくなりすぎ、ユーザーへの実質的な影響はほとんどないため、推奨されない。
- **1 秒あたりのトランザクション数。**これにより、システムでどれだけの作業が行われたかの情報が得られる（システムのスループット）。
- **トランザクション故障数。** このデータは、1 秒あたりのトランザクション数を分析する際に使用される。故障とは、期待されたイベントやプロセスが完了しなかった、または実行されなかったことを示す。故障が発生した場合は懸念材料となり、根本的な原因を調査する必要がある。失敗したトランザクションは、完了したトランザクションよりもはるかに短い時間しか要しないため、失敗したトランザクションによって、1 秒あたりのトランザクション数のデータが無効になることもある。
- **1 秒あたりのヒット数（またはリクエスト数）。**これにより、テストの各秒間にシミュレーションされたユーザーがサーバーにアクセスした回数を知ることができる。
- **ネットワークのスループット。**これは通常、1 秒あたりのビット数のように、時間間隔によるビット数で測定される。これは、シミュレーションされたユーザーが 1 秒ごとにサーバーから受け取るデータ量を表している。(4.2.5 項参照)
- **HTTP レスポンス。**これらは 1 秒ごとに測定され、200, 302, 304, 404 のようなレスポンスコードを含む。404 はページが見つからないことを示す。

これらの情報の多くは表で表現できるが、グラフで表現することにより、データを見やすく、傾向を把握しやすくできる。

データの分析に使う技法には、以下のものがある。

- 要件に対する結果の比較
- 結果の傾向を観測
- 統計的品質管理手法
- エラーの確認
- 期待値と実績の比較
- 過去のテスト結果との比較
- コンポーネント（サーバー、ネットワークなど）の正常に機能していることの検証

メトリクス間の相関関係を把握すると、システムの性能がどの時点で低下し始めるかを理解することに役立つ。例えば、CPU がキャパシティの 90%に達してシステムがスローダウンしたとき、1 秒間に何件のトランザクションが処理されていたか？

分析は、性能低下や故障の根本原因を特定し、それを修正することに役立てることができる。確認テストは、修正アクションが根本原因を解決したか否かの判断に役立つ。

レポート

分析結果は、集約し、性能テスト計画書に記載された目的と比較する。これらは、他のテスト結果と一緒に全体的なテストステータスレポートとして報告するか、または性能テスト専用のレポートとして報告する。レポートの詳細レベルは、ステークホルダーのニーズに合わせるべきである。これらの結果に基づく推奨事項は、通常、ソフトウェアのリリース基準（ターゲット環境を含む）や必要な性能の改善に対応するものである。

典型的な性能テストのレポートには以下の内容が含まれる。

エグゼクティブサマリー

この節は、すべての性能テストが行われ、すべての結果が分析され、理解された後に完成する。この節のゴールは、簡潔で理解しやすい結論、所見、およびマネジメントへの提言を、提言による成果目標と一緒に提示することである。

テスト結果

テスト結果には、以下の情報の一部またはすべてが含まれる。

- 結果として提供する説明や詳細のサマリー。
- ある時点でのシステム性能の「スナップショット」としての役割を果たし、後続のテストとの比較の基礎となるベースラインテストの結果。テスト結果には、テストを開始した日時、同時使用ユーザーの目標値、計測したスループット、および主な所見が含まれる。主な所見には、計測した全体的なエラーレート、応答時間、平均スループットを含む。
- テストの目的に影響を与える可能性のある（または与えた）アーキテクチャーコンポーネントを示す概略図。
- 応答時間、トランザクションレート、エラーレート、性能分析を示すテスト結果の詳細な分析（表やグラフ）。分析結果には、安定していたアプリケーションがどの時点で不安定になったか、故障の原因（Web サーバー、データベースサーバーなど）など、観測された内容の説明も含む。

テスト結果記録/情報の記録

各テストケースの動作ログの記録を残すべきである。ログには通常、以下の内容が含まれる。

- テスト開始の日付・時刻
- テスト期間
- テストに使用されたスクリプト（複数のスクリプトを使用している場合は混合したスクリプトを含む）および関連するスクリプト構成データ
- テストで使用されるテストデータファイル（複数可）
- テスト中に作成されたデータ/ログファイルの名前と場所
- テストされた HW/SW 構成（特に複数のテスト実行間の変更点）
- Web サーバーおよびデータベースサーバーの CPU および RAM の平均およびピーク使用率
- 達成した性能に関する注記
- 識別された欠陥

推奨事項

テストの結果から得られる推奨事項には以下の内容が含まれる。

- ハードウェアやソフトウェア、ネットワークインフラの再構成など、推奨される技術的な変更
- 更なる分析のために特定された領域（例：問題やエラーの根本原因を特定するための Web サーバーログの分析など）

- 性能の特性や傾向（劣化など）を測定するためのより詳細なデータを得るために必要なゲートウェイ、サーバー、ネットワークの追加のモニタリング

5. ツール – 90 分

キーワード
ロードジェネレーター、負荷マネジメント、モニタリングツール、性能テストツール

「ツール」の学習の目的

5.1 ツールサポート
PTFL-5.1.1 (K2) ツールがどのように性能テストをサポートするか理解する。

5.2 ツールの適合性
PTFL-5.2.1 (K4) 与えられたプロジェクトシナリオにおいて、性能テストツールの適合性を評価することができる。

5.1 ツールサポート

性能テストをサポートするツールには、次のような種類がある。

ロードジェネレーター
ジェネレーターは、IDE、スクリプトエディター、またはツールスイートを通じて、定義された運用プロファイルに従ってユーザーの行動をシミュレーションする複数のクライアントインスタンスを作成し、実行することができる。短時間に複数のインスタンスを作成すると、テスト対象のシステムに負荷がかかる。ジェネレーターは、負荷を発生させるとともに、後で報告するためのメトリクスを収集する。

性能テストケースを実行する際のロードジェネレーターの目的は、現実の世界をできる限り模倣することである。これは、多くの場合、テストを行う場所だけでなく、さまざまな場所からユーザーリクエストを送ることが必要になることを意味する。複数のポイントオブプレゼンスを設定した環境では、負荷の発生源が分散されるため、すべてが単一のネットワークから発生することはない。これにより、テストにリアリティを与えることができる。ただし、ネットワーク上の複数のホップで遅延が発生すると、結果が歪むことがある。

負荷マネジメントコンソール
負荷マネジメントコンソールは、ロードジェネレーターの起動と停止をコントロールする。また、このコンソールは、ジェネレーターが使用するロードインスタンス

内で定義されているさまざまなトランザクションからのメトリクスを集約する。このコンソールでは、テスト実行からのレポートやグラフを見ることができ、結果分析をサポートする。

モニタリングツール
モニタリングツールは、テスト対象のコンポーネントやシステムと同時に動作し、コンポーネントやシステムの動作をモニタリング、記録、そして／または分析する。典型的には、Web サーバーのキュー、システムメモリー、ディスクスペースなどがモニタリング対象となる。モニタリングツールは、テスト対象システムにおける性能低下の根本原因の分析を効果的にサポートし、また、製品がリリースされた際の本番環境のモニタリングにも使用できる。性能テストの実行中、モニターはロードジェネレーター自体にも使用されることがある。

性能テストツールのライセンスモデルには、完全な所有権を持つ従来のシート／サイトに基づいたライセンス、クラウドベースの従量課金ライセンスモデル、定義された環境またはクラウドベースのオファリングを通じて無料で使用できるオープンソースライセンスなどがある。それぞれのモデルは、異なるコスト構造を意味し、継続的なメンテナンスを含む場合もある。どのツールを選択するにしても、そのツールがどのように機能するかを（トレーニングそして／または自習を通じて）理解するには、時間と予算が必要であることは確かである。

5.2 ツールの適合性

性能テストツールを選択する際には、以下の要因を考慮する必要がある。

互換性
一般的にツールは、単独のプロジェクトのためだけでなく、組織のために選択される。これは、組織における以下の要因を考慮することを意味する。

- プロトコル：4.2.1 項で述べたように、プロトコルは性能ツールの選択において非常に重要な側面である。システムがどのプロトコルを使用しているか、また、どのプロトコルをテストするかを理解することは、適切なテストツールを評価するために必要な情報となる。
- 外部コンポーネントとのインターフェース：ソフトウェアコンポーネントまたは他のツールとのインターフェースは、プロセスまたは他の相互運用性の要件（例えば、CI プロセスにおける統合）を満たすために、完全な統合要件の一部として考慮が必要な場合がある。

- プラットフォーム：組織内のプラットフォーム（およびそのバージョン）との互換性は不可欠である。これは、ツールをホストするためのプラットフォームと、モニタリングそして／または負荷生成のためにツールがやりとりするプラットフォームに適用する。

拡張性

考慮すべき別の要因として、そのツールが扱える同時ユーザーシミュレーションの総数がある。これにはいくつかの要因が含まれる。

- 最大必要ライセンス数
- 負荷生成をするワークステーション/サーバーの構成要件
- 複数の拠点から負荷を生成する能力（例：分散サーバー）

理解性

もう 1 つの考慮すべき要因は、ツールを使用するために必要な技術的知識のレベルである。この要因は見落とされがちで、技術を有しないテスト担当者が誤ったテスト設定を行い、不正確な結果をもたらす可能性がある。複雑なシナリオや高度なプログラミング、カスタマイズを必要とするテストの場合、チームはテスターが必要なスキル、バックグラウンドを有し、トレーニングを受けていることを確認すべきである。

モニタリング

ツールが提供するモニタリングは十分か？そのツールによるモニタリングを補完するために、環境内で利用可能な他のモニタリングツールはあるか？定義したトランザクションにモニタリングを関連付けることができるか？プロジェクトで必要とされるモニタリングをツールが提供できるかどうかを判断するためには、これらすべての質問に答えなければならない。

モニタリングが独立したプログラム/ツール/ホールスタック(whole stack)である場合は、プロダクトがリリースされたときに本番環境をモニタリングするために使用することができる。

6. 参考文献

6.1 標準

- [ISO25000] ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation（SQuaRE）

6.2 ISTQB®ドキュメント

- [ISTQB_UT_SYL] ISTQB® Foundation Level Usability Testing Syllabus, Version 2018
- [ISTQB_ALTA_SYL] ISTQB® Advanced Level Test Analyst Syllabus, Version 2012
  JSTQB 訳注）日本では「ISTQB® テスト技術者資格制度 Advanced Level シラバス日本語版 テストアナリスト Version 3.1.1.J03」として発行されている。
- [ISTQB_ALTTA_SYL] ISTQB® Advanced Level Technical Test Analyst Syllabus, Version2012
  JSTQB 訳注）日本では「テスト技術者資格制度 Advanced Level シラバス日本語版 テクニカルテストアナリスト Version 2012.J02」として発行されている。
- [ISTQB_ALTM_SYL] ISTQB® Advanced Level Test Manager Syllabus, Version 2012
  JSTQB 訳注）日本では「テスト技術者資格制度 Advanced Level シラバス日本語版 テストマネージャ Version 2012.J04」として発行されている。
- [ISTQB_FL_SYL] ISTQB® Foundation Level (Core) Syllabus, Version 2018
- JSTQB 訳注）日本では「テスト技術者資格制度 Foundation Level シラバス Version 2018V3.1.J03」として発行されている。
- [ISTQB_FL_AT] ISTQB® Foundation Level Agile Tester Syllabus, Version 2014
- JSTQB 訳注）日本では「テスト技術者資格制度 Foundation Level Extension シラバス アジャイルテスト担当者 Version 2014.J01」として発行されている。
- [ISTQB_GLOSSARY] ISTQB® Glossary of Terms used in Software Testing, https://glossary.istqb.org

6.3 書籍

[Anderson01] Lorin W. Anderson, David R. Krathwohl (eds.) "A Taxonomy for Learning, Teaching and Assessing:A Taxonomy for Learning, Teaching and Assessing: A

Revision of Bloom's Taxonomy of Educational Objectives", Allyn & Bacon, 2001, ISBN 978-0801319037

[Bath14] Graham Bath, Judy McKay, "The Software Test Engineer's Handbook", Rocky Nook, 2014, ISBN 978-1-933952-24-6

[Molyneaux09] Ian Molyneaux, "The Art of Application Performance Testing:From Strategy to Tools", O'Reilly, 2009, ISBN: 9780596520663

JSTQB 訳注）日本では「アート・オブ・アプリケーション パフォーマンステスト」(オライリージャパン，2009 年)として発行されている。

[Microsoft07] Microsoft Corporation, "Performance Testing Guidance for Web Applications", Microsoft, 2007, ISBN: 9780735625709